UNIVERSITY of CALIFORNIA
Santa Barbara

Data Model Verification via Theorem Proving

A Dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Ivan Bocić

Committee in Charge:

Professor Tevfik Bultan, Chair
Professor Ben Hardekopf
Professor Jianwen Su

December 2016

The Dissertation of Ivan Bocić is approved.

Professor Ben Hardekopf

Professor Jianwen Su

Professor Tevfik Bultan, Committee Chair

September 2016

Curriculum Vitæ

Ivan Bocić

Education

2016	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2015	M.A. in Computer Science, University of California, Santa Barbara.
2010	B.Sc. in Computer Science, School of Computing at Union University, Belgrade, Serbia.

Publications

ASE 2016	Ivan Bocić, Tevfik Bultan. "Finding Access Control Bugs in Web Applications with CanCheck"
ASE 2015	Ivan Bocić, Tevfik Bultan. "Efficient Data Model Verification with Many-Sorted Logic"
ICSE 2015	Ivan Bocić, Tevfik Bultan. "Coexecutability for Efficient Verification of Data Model Updates"
NFM 2015	Ivan Bocić, Tevfik Bultan. "Data Model Bugs"
TOSEM 2015	Jaideep Nijjar, Ivan Bocić, Tevfik Bultan. "Data Model Property Inference, Verification and Repair for Web Applications"
GSWC 2014	Ivan Bocić, Tevfik Bultan. "Coexecutability: How To Automatically Verify Loops"
ICSE 2014	Ivan Bocić, Tevfik Bultan. "Inductive Verification of Data Model Invariants for Web Applications"
FormaliSE 2013	Jaideep Nijjar, Ivan Bocić, Tevfik Bultan. "An Integrated Data Model Verifier with Property Templates"

Abstract

Data Model Verification via Theorem Proving

by

Ivan Bocić

Software applications have moved from desktop computers onto the web. This is not surprising since there are many advantages that web applications provide, such as ubiquitous access and distributed processing power. However, these benefits come at a cost. Web applications are complex distributed systems written in multiple languages. As such, they are prone to errors at any stage of development, and difficult to verify, or even test. Considering that web applications store and manage data for millions (even billions) of users, errors in web applications can have disastrous effects.

In this dissertation, we present a method for verifying code that is used to access and modify data in web applications. We focus on applications that use frameworks such as Ruby on Rails, Django or Spring. These frameworks are RESTful, enforce the Model-View-Controller architecture, and use Object Relational Mapping libraries to manipulate data. We developed a formal model for data stores and data store manipulation, including access control. We developed a translation of these models to formulas in First Order Logic (FOL) that allows for verification of data model invariants using off-the-shelf FOL theorem provers. In addition, we developed a method for extracting these models from existing applications implemented in Ruby on Rails. Our results demonstrate that our approach is applicable to real world applications, it is able to discover previously unknown bugs, and it does so within minutes on commonly available hardware.

Contents

Curriculum Vitae **iii**

Abstract **iv**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**
- 1.1 Data Model Bug Examples . . . 3
- 1.2 Discussion on Data Model Bugs . . . 7
- 1.3 Our Approach Overview . . . 11
- 1.4 Contributions . . . 15

2 Abstract Data Stores **17**
- 2.1 Data Models in Ruby on Rails . . . 17
- 2.2 Abstract Data Stores . . . 21
- 2.3 Data Store Correctness . . . 27
- 2.4 Abstract Data Store Language . . . 31

3 Model Extraction **35**
- 3.1 Model Extraction and Dynamic Features of Ruby . . . 36
- 3.2 Symbolic Model Extraction . . . 39
- 3.3 Symbolic Extraction for Data Model Verification . . . 51
- 3.4 Experiments . . . 59

4 Verification via First Order Logic **63**
- 4.1 Classical First Order Logic . . . 63
- 4.2 ADS Translation . . . 65
- 4.3 Experimental Evaluation . . . 77

5 Coexecutability **83**
5.1 Coexecution overview 84
5.2 Formalization 90
5.3 Syntactic Analysis 101
5.4 Experimental Evaluation 106

6 Verification via Many-sorted Logic **110**
6.1 Many-Sorted Logic 111
6.2 Empty Logic 112
6.3 Translation to Many-sorted Logic 114
6.4 Experimental Evaluation 120

7 Related Work **129**
7.1 Modeling and Verification of Web Applications 129
7.2 Access Control 133
7.3 Theorem Prover Based Verification 134
7.4 Coexecution 136
7.5 Extraction 137

8 Conclusion **140**

Bibliography **143**

List of Figures

1.1 Architecture imposed by web application frameworks 2
1.2 Verification of data model invariants 12

2.1 Excerpt from a Rails application 18
2.2 Class diagram corresponding to Figure 2.1 19
2.3 Action translation example 25

3.1 Example of Rails dynamic features 36
3.2 Static equivalent to action in Figure 3.1. 37
3.3 Model extracted from Figure 3.1. 39
3.4 Overview of symbolic model extraction 41
3.5 Symbolic model extraction from dynamically generated methods. 45
3.6 Symbolic model extraction example. 48
3.7 Ability class from Figure 2.1 54
3.8 Model extracted from the authorization check in line 48 of Figure 3.7. . . 59

4.1 A data model schema example based on FatFreeCRM [36] 66
4.2 Axioms defining the class diagram in Figure 4.1 in classical FOL 67

5.1 An example action 84
5.2 Example of sequential execution 85
5.3 Example of coexecution 86
5.4 Unparallelizable but coexecutable loop 88
5.5 Parallelizable but not coexecutable loop 89
5.6 Sequential execution vs. coexecution 94
5.7 Syntactic analysis pseudocode 103
5.8 Application information and verification results 108

6.1 A data model schema example based on FatFreeCRM [36] 114
6.2 Axioms defining the class diagram in Figure 6.1 in classical (unsorted) first order logic 115
6.3 Axioms defining the class diagram in Figure 6.1 in many-sorted logic . . 116

6.4 Example action based on FatFreeCRM [36] 118
6.5 Unsorted action translation example 118
6.6 Many-sorted action translation example 119
6.7 Verification time distribution 122
6.8 Distribution of the slowdown factor compared to (many-sorted) Z3 124

List of Tables

2.1 State-migrating abstract data store statement nodes 32
2.2 State-preserving abstract data store statement nodes 33

3.1 Instrumentation for symbolic extraction 43
3.2 ActiveRecord methods and corresponding ADS statement nodes 52
3.3 Mapping actions to CRUD operations 55
3.4 Experimental results for symbolic extraction 60

4.1 Verification experiments summary 78

6.1 Verification performance summary 122
6.2 Observed slowdown compared to (many-sorted) Z3 124
6.3 Coexecution vs sequential execution with many-sorted logic 128

Chapter 1

Introduction

Nowadays, most computers are connected to the Internet. This network connectivity, when combined with the increasingly prevalent cloud computing platforms, enables software applications to store data on remote servers and use thin clients (web browsers or mobile applications) that provide access to application data from any device, anywhere, anytime, without maintaining any local copies. Web applications have started to play a significant role in improving the efficiency of national infrastructures in many critical areas such as healthcare [2, 49], policy making [105], national security, and the power grid [50].

However, these applications are challenging to develop and maintain since they are complex software systems consisting of distributed components that run concurrently and interact over the Internet. In order to reduce this complexity and achieve modularity, web application frameworks have been developed for various languages: Ruby on Rails [87] for Ruby, Django [31] for Python, and Spring [94] for Java. These frameworks, while developed for different language, share similar architectures and basic features, as shown in high level in Figure 1.1.

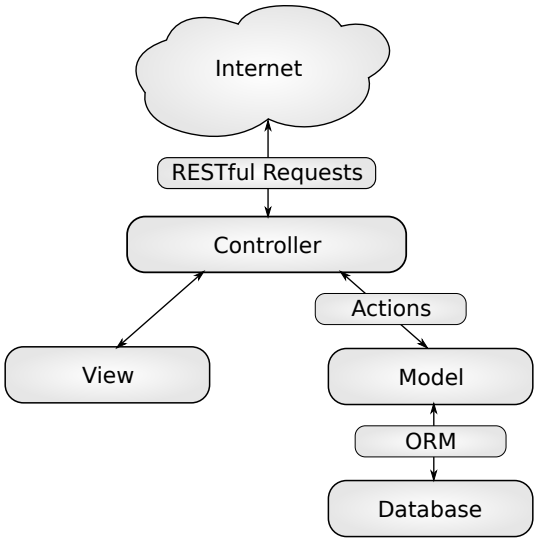


Figure 1.1: Architecture imposed by web application frameworks

These web application development frameworks use the Model-View-Controller (MVC) pattern [69] to separate the code for the model (Model) from the user interface logic (View) and the navigation logic (Controller).

The model's responsibility is to define all the data that the web application manages and stores in the persistent data store. Typically, the model is implemented using Object-Relational Mapping (ORM) libraries. ORM libraries require that the ORM schema is configured. This configuration encompasses declaring the classes of the model, basic type fields of these classes, and special declarations for all associations between these classes. This configuration is loaded at runtime and serves to guide the execution of ORM library commands.

Actions are defined by the controller as operations that the user can invoke to browse and/or modify the data. A widely used paradigm in web application frameworks is the Representational State Transfer (REST) architecture with RESTful interfaces. In RESTful applications, any action on the data model can be invoked at any time and any number of times. These actions should be atomic and execute quickly. All requests

should provide all the information needed to fulfil the request. RESTful applications are supposed to be scalable, reliable and performant [39].

None of the major web application frameworks support access control by default. Instead, the access control implementation is left to the developers and third party libraries. Access control is typically enforced at runtime: inside an action, there will typically be an authorization check. If this check fails, the action will abort without modifying or reading any data. In addition, access control is typically role-based: each user is assigned a role, and roles define user access. However, since these checks are either written manually or automatically generated using heuristics, the enforcement of access control may be lacking.

For data-oriented web applications, the correctness of actions that update the data store is the most significant correctness concern since erroneous actions can lead to corruption, loss or theft of data.

1.1 Data Model Bug Examples

Data model bugs are bugs related to data storage and manipulation in web applications [14]. In this section we analyze data model bugs that were found in several real world Ruby on Rails applications by automated data model verification techniques. We discuss the nature of these bugs: their severity, potential for discovery, recovery and prevention.

Before we proceed to discuss data model bugs in general, we will list four web applications and show examples of data model bugs that were found in them [13, 78, 75, 16]. These bugs vary in nature, severity and potential for recovery, serving as useful background for a deeper discussion.

FatFreeCRM[1] is an application for customer-relation management. It allows for storing and managing customer data, leads that may potentially become customers, contacts, campaigns for marketing etc. The code is written in a highly dynamic and reusable way, with different controllers running the same source code, dynamically loading and referring to specific classes based on the circumstances. It spans 20178 lines of Ruby code, 32 model classes and 120 actions. Using data integrity verification [12], we found two bugs in FatFreeCRM that we reported to the developers, who confirmed them and immediately fixed one of them. In future discussion we refer to these two bugs as F1 and F2. Rubicon [75] is a tool for verification of Ruby of Rails applications that translates abstract unit tests to Alloy [62], with the goal of ensuring that these tests would pass when given any set of concrete objects. Bug F3, related to access control, was detected using Rubicon.

Bug F1 is caused by `Todo` objects, normally associated with a specific `User`, to not get deleted when their `User` is deleted. We call these `Todo` objects *orphaned*. Orphaned `Todo` objects are fundamentally invalid because the application assumes that their owner exists, causing crashes whenever an orphaned `Todo`'s owner is accessed. Because of the severity, this bug was acknowledged and repaired immediately after we submitted a bug report. It is interesting to note that the same bug could not be replicated upon deleting other types of objects that belong to a `User`, only `Todo`s were not properly cleaned up.

Bug F2 relates to `Permission` objects. `Permission` objects serve to define access permissions for either a `User` or a `Group` to a given `Asset`. Our tool has found that it is possible to have a `Permission` without associated `User` or `Group` objects. This bug is replicated by deleting a `Group` that has associated `Permission`s. Interestingly, this bug cannot be replicated by deleting `Permission`s of a `User`, only `Permission`s of `Group`s. Although similar to F1 in causality, the repercussions of this bugs are very different. If there exists an `Asset`

[1] www.fatfreecrm.com

object whose all `Permission` objects do not have associated `Users` or `Groups`, it is possible to expose these assets to the public without any user receiving an error message, and without any `User` or `Group` owning and managing this asset.

Bug F3 is an access control bug that exposes a `User`'s private `Opportunity` objects to other `Users`. This bug is exploited by registering a new `User` in a way that it shares some of the target `User`'s `Contacts`, giving access to private `Opportunity` objects through these `Contacts`. This bug was caused by a false assumption by the developers that all `Opportunity` and `Contact` objects that belong to the same person will have the same `Permissions`. This bug was reported and acknowledged by the developers.

Tracks[2] is an application for organizing tasks, to-do lists etc. This application spans 17562 lines of code, 11 model classes and 117 actions. We identify four bugs in Tracks, which we refer to as T1, T2, T3, and T4. Bug T3 was detected using data model schema verification [78]. Bugs T1, T2 and T4 were discovered using data integrity verification [12], were reported to and have been fixed by the developers.

Bug T1 is related to the possibility of orphaning an instance of a `Dependent` class. Deleting a `Todo` directly, using the action dedicated to deleting `Todos`, cleans up related `Dependent` objects properly. However, The action that deletes a `Project` also deletes all `Todos` of that `Project` without deleting the `Dependent` objects of deleted `Todos`. Since only `Todos` of the same project can be interdependent, the orphaned `Dependent` objects are completely disconnected from any remaining `Todos`. This bug is similar to bugs F1 and F2, except that the orphaned objects cannot be accessed by the user in any way. Therefore, this bug does not affect the semantics of the application. However, it does present a memory-leak like bug, affecting performance by unnecessarily populating database tables and indexes.

[2] getontracks.org

Bug T2 is very similar in nature to T1. When a `User` is deleted, all `Projects` of the `User` are deleted as well, but `Notes` of deleted `Projects` remain orphaned. These orphaned `Note` objects are not accessible in any way, however, the orphaned `Todos` take up space in the database and inflate indexes.

Bug T3 is caused by deleting a `Context` without correctly cleaning up associated `RecurringTodo` objects. This is similar to bug F1 because the orphaned `RecurringTodo` objects are accessible by the application and cause the application to crash.

We found bug T4 when the action verification method [12] reported an inconclusive result within the action used to create `Dependent` instances between two given `Todos`. Semantically, there must not be dependency cycles between `Todos`; this is a structural property of the application. Our method could not prove or disprove that cycles between `Todos` cannot be created. Upon manual inspection we found that, while the UI prevents this, HTTP requests can be made to create a cycle between `Todos`. The repercussions of this bug are potentially enormous. Whenever the application traverses the predecessor list of a `Todo` inside a dependency cycle it will get stuck in an infinite loop, eventually crashing the thread and posting an error to the `User`. No error is shown when the user creates this cycle, only later upon accessing it. This creates a situation when repairing the state of the data may be impossible.

LovdByLess[3] is a social networking application. It allows people to create accounts, write posts and comment on posts of other users, upload and share images etc. It contains 29667 lines of code, 12 model classes and 100 actions.

Data model schema analysis [78] was used to find bug L1 where the `Comments` of a `User` were not cleaned up properly when a `User` is deleted. The orphaned `Comments`, however, remain connected to the `Post` they belong to, and are visible from the said `Post`. The

[3] github.com/stevenbristol/lovd-by-less

application previews these `Comments`, along with their content and other data, except for the author. The author's name field remains blank. This is not expected behavior: either the `Comments` are supposed to be deleted, or they are supposed to remain, in which case the author's data is lost.

CoRM[4] is a consumer relationship manager designed for small businesses. It spans 7745 lines of Ruby code, with 39 model classes and 163 actions. We found more than one bug in this application [16], but here we will present a single bug C1. While the admin panel is not accessible to non-administrators, a specific page that is used for batch importing and exporting data (related to the `ImportsController`) is accessible by all users. This is due to a lack of an access control check. While normal users will never see the link needed to access this panel, experienced Rails developers can target the controller directly and maliciously read or alter any data used by the application.

1.2 Discussion on Data Model Bugs

We identified two types of bugs: access control bugs and data integrity bugs. Access control bugs give access to data to users with insufficient privileges. Data integrity bugs are bugs that allow invalidation of the application's data. Note that we draw a distinction between bugs that allow data to be invalidated and bugs that are caused by data that has been invalidated. The latter bug is a symptom of the former.

1.2.1 Severity

Access control policies are hard to correctly specify and hard to correctly enforce [75]. Access control bugs are severe bugs. Exposing private information is not permissible in

[4] https://github.com/SIGIRE/CoRM

any application that stores and manages private information, nor is allowing access to admin or root level operations. This is especially true for bug C1, where any user can maliciously control all of the applications data.

The severity of data integrity bugs varies on the specifics of the bug, spanning from benign bugs that at most cause minor performance problems, over bugs causing crashes in the application, to bugs causing data loss and corruption from which recovery is exceptionally difficult or impossible.

We identified several data integrity bugs that allow invalid data to exist in the database, but in such a way that this invalid data is never used by the application. We refer to these bugs as *data model leaks*. They are usually caused by incorrect cleanup of related entities when an entity is deleted. This category is demonstrated by bugs T1 and T2. These bugs are hard to detect unless the leaked data accumulates to a certain point. Their impact is limited to performance, not affecting the semantics of the program. They negatively impact performance by taking up space in the database and populating indexes unnecessarily.

In many cases, corrupted data can be accessed by the application, causing the application to misbehave in some way. We identified a wide range of misbehavior severity. For example, orphaned objects may be visible to the user as empty fields on the webpage (L1), allow operations and further data updates that should not be allowed (F2), or crash the web application (F1, T3, T4).

1.2.2 Recovery

Access control bugs allow no recovery. Once private information has been exposed, fixing the bug only prevents future threats. No measure exists to make the exposed information private again. Furthermore, in case of bug C1, since the malicious user

can modify the applications data in any way, graceful recovery may be impossible and backups would have to be used to restore data. Data integrity bugs generally have a higher recovery potential. Repairing a data integrity bug involves two steps: repairing the data and preventing future invalidation.

In some cases, data is recoverable. For example, once a data model leak is discovered, leaked entities can be identified and removed. The same applies in the case of data being incorrectly deleted: bugs F1, F2, T3 are recoverable from because the original intent of the developer was to delete data. Removing the invalid data not only removes the corruption, but also brings the data store to the state that was originally expected by the developers.

Data integrity bugs that do not manifest themselves through improper deletion are far more difficult to recover from. Repairing the corruption implies modifying the corrupted data into valid data, which may be impossible. T4 is an example of a bug in which valid data is not distinguishable from invalid data. Even clearly distinguishable corrupted data may be unrecoverable if, for example, invalid data has overwritten correct data and the corrupted data seeped into long term backups.

Backups can be used to recover corrupted data in certain cases. This would be a manual and error prone effort, however, and it would rollback the user's data to a previous point which may be undesirable. To make matters worse, since data integrity bugs are observable only if the data has already been invalidated, the corruption may have been backed up in the time frame between the cause of the corruption and the escalation of the bug, making short term backups unusable.

1.2.3 Detection and Prevention

Data model bugs are hard to anticipate, and addressing them after being detected by users is undesirable because recovery may be extremely difficult. Detection of access control bugs after they are exposed is difficult since a malicious user may leave no trace when accessing restricted information. Similarly, data integrity bugs are hard to detect. They are not observed until the application accesses the invalidated (corrupted) data and misbehaves, which may not be possible (as is the case with bugs T1 and T2). If a user does access the invalidated data, the resulting faulty behavior cannot be replicated by the developer without being given access to the same invalidated data. Furthermore, even given access to this data, the code causing this strange behavior may be correct. No trace might exist on how the data was originally invalidated.

Runtime validation is a commonly used technique for the prevention of potential data model integrity bugs. We define runtime validation as any runtime check that aborts the operation with the goal of preventing invalidation. In web application frameworks, validation can be done in the web application layer automatically (both Rails and Django support user definable model validators), or could be manually implemented in actions (in form of conditional branches that abort unless a specific condition is met), or in the database by defining constraints. Frequently multiple approaches are used: for example, the database may validate the integrity of foreign keys, whereas the application layer may validate that email strings adhere to a given format. Runtime validation alone provides an insufficient solution to the problem. This is demonstrated by the fact that we found serious bugs in applications that rely heavily on runtime validation, and attempted to enforce correctness and security policies. For all the bugs we found, the problem was caused by incorrect implementation of runtime validation.

1.3 Our Approach Overview

Considering the difficulty of detection, potential severity and unrecoverability of data model bugs, we strongly believe that automated verification techniques should be used to prevent data model bugs. However, given the complexity and distributed nature of web applications, static analysis is rarely, if ever, used to address the correctness concerns developers might have. Instead, testing has become the industry standard. While writing tests requires little developer training and quickly finds defects covered by the test suite, complete assurance of correctness is practically impossible. Behaviors not covered by the test suite could always contain bugs, and high coverage is difficult to achieve.

Static verification allows a rigorous method of analyzing an application systematically and exhaustively. However, the thoroughness of static verification comes at a cost. Static verification techniques either have problem scaling to real world applications, require manual effort by developers with extensive training and experience, or risk reporting false positives doe to coarse abstractions. As such, static verification is rarely, if ever, used in real world application development.

The focus of our work is the verification of data models and data model behaviors in web applications developed using web application frameworks. We posit that the conventions and modularity that these frameworks impose make static verification of data models feasible in practice and useful for finding bugs with little developer effort.

Our key observation is that, in RESTful applications, actions are (or should be) atomic and can be executed in any order. This lets us use inductive verification to verify data integrity by considering each action in isolation and checking whether an action could possibly invalidate a property that was presumed to be valid before the action executed. If no action breaks any property, then assuming the application starts executing in a valid state, no invalid state could possibly be reached.

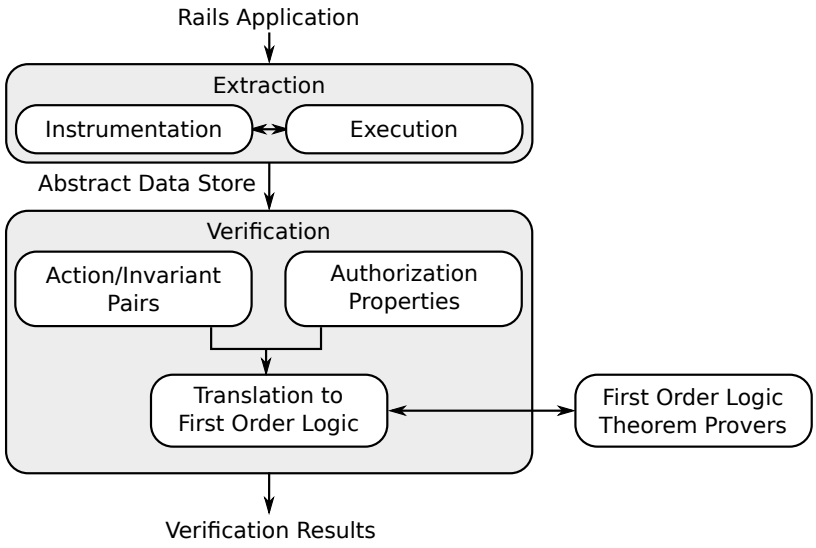


Figure 1.2: Verification of data model invariants

Besides reasoning about how actions modify data with regards to data integrity, we can reason about how they modify data with regards to access control. This gives us power to verify access control statically, to ensure that access control enforcement is correct with regards to the desirable access control policy. Specifically, we can identify *authorization properties*: signatures of how an application modifies or exposes data depending on the role of the current user. These signatures can be automatically verified against the access control policy that a developer already wrote for a third party authorization library.

We developed a novel approach for automatically verifying data integrity and access control for web applications (Figure 1.2). First, by exploiting the structure of the MVC-pattern, we automatically extract an abstract specification of the data model, called *abstract data store*. Abstract data stores include information on the database schema, actions that update the data store, as well as access control information. Next, we convert verification queries about the data model (stated as invariants) to formulas in First-Order-Logic (FOL) based on inductive invariant verification in order to verify data integrity.

In addition, we identify authorization properties in order to verify access control, which we also translate to FOL. Finally, we use automated FOL theorem provers to verify data integrity and access control.

We implemented our approach for Ruby on Rails [87], or Rails in short. We decided to focus on the Rails framework since it is widely used, however, our approach can be adapted to other MVC-based web application frameworks such as Django [31] and Spring [94].

We developed a Rails library that lets developers verify data integrity or access control in their Rails applications. To verify data integrity, developers need to specify desirable properties, or invariants, they expect to hold in their web application. To verify access control enforcement, the developer needs to utilize CanCan, a popular third party access control library, to implement access control. Although our implementation supports CanCan only, this is not a limitation of our general approach.

Given a Rails application and a set of invariants and/or a compatible access control system, the verification approach consists of two major phases. The first phase is automatic extraction of a formal specification, which we call an *Abstract Data Store (ADS)*, that characterizes the model and the actions of the input web application, as well as invariants and access control information if applicable. ADS models the data store as sets of objects (corresponding to objects of the data model classes) and associations among them (corresponding to the associations among the data model classes). Attributes that correspond to basic types are not modeled (i.e., they are abstracted away). This means that we can verify invariants about sets of objects and associations among them (like the example above), but for example, not about numeric attributes of objects.

The crucial part of the extraction phase is extraction of action specifications, where actions of the web applications are translated to ADS actions. ADS actions contain constructs for creating and deleting objects and updating associations, and they allow

non-determinism, which is necessary due to abstraction of the attributes with basic types.

We exploit the MVC-pattern during the extraction phase. Actions that update the data model correspond to actions that are executed in response to user requests. We can ignore the View construction since it does not influence the data store state. All other parts of the application are irrelevant for our purposes.

Statically extracting actions specifications from Rails code is still challenging due to the dynamic nature of the Ruby language. To address this challenge, we developed a novel model extraction technique for programs written using dynamically typed languages. We extract the model of the entire application in a single execution of the instrumented application in a way that avoids path explosion. Furthermore, we implement program instrumentation in the source program language itself, Ruby in our case, in order to be able to instrument and extract models from dynamically generated methods.

The second major phase of our approach is automated verification of inductive invariants on the extracted model using First Order Logic (FOL) theorem provers. For each action/invariant pair we generate a FOL theorem that checks whether, assuming all invariants hold in the action's pre-state, the action preserves the invariant. If there exists an access control policy, we also generate FOL theorems that ensure that any operations done by an action conform to the access control policy in all cases.

These FOL theorems are sent to a theorem prover to check if they are correct (which means that the invariant holds, or access control is correctly enforced), or incorrect (which means that the invariant or the access control policy can be violated by the action). Alternatively, since our translation produces arbitrarily nested quantification and FOL is undecidable in general, the theorem prover may not produce a conclusive result. As such, if a theorem prover does not deduce a result within a time frame, we stop the deduction and mark the result as *inconclusive*. Inconclusive results are supposed to be followed up by manual investigation by the developer, and as such, do not provide

useful feedback to the developer. As a consequence of our efforts to reduce the rate of inconclusive results (Chapters 5 and 6), we managed to reduce the rate of inconclusive results to 0.14% over a number of real world Ruby on Rails applications.

1.4 Contributions

The contributions of this dissertation are:

- The Abstract Data Store (ADS) modeling language, that represents portions of web applications that are focusing on the data model schema, actions, as invariants and access control information,
- A way to translate ADS models to classical first order logic for verification of data integrity and access control,
- A method for model extraction from web applications written in dynamically typed programming languages,
- An approach for translating loops to FOL that is equivalent to modeling sequential sequences of iterations, but generally easier to verify, and
- Discussions and experimental evaluations on translating ADS models to different kinds of non-classical first order logic and how they can be leveraged to increase verification viability.

The rest of this dissertation is organized as follows. Chapter 2 defines Abstract Data Stores, a model of the data types web applications manipulate, as well as the ways in which this data is being manipulated. Chapter 3 discusses the extraction method we implemented to extract ADSs from Rails applications. Chapter 4 defines our basic approach to verification of ADSs [12]. Chapter 5 introduces an alternate way of modeling

loops in data model verification [13]. Chapter 6 discusses non-classical FOL and how the translation can be modified to support non-classical FOL and other theorem provers [15]. Chapter 7 discusses related work, and Chapter 8 concludes the dissertation.

Chapter 2

Abstract Data Stores

In this chapter we define abstract data stores (ADSs). After presenting an example Rails data model, we will introduce formalisms that define the data a web application is designed to store and manage, as well as the ways in which this data is being managed. In addition, ADSs can have expected properties (invariants) defined on them, or have an access control policy.

2.1 Data Models in Ruby on Rails

Figure 2.1 presents an example of an excerpt of a Rails application based on Tracks [103]. This excerpt would normally be contained in multiple files, one for each model class and one for each controller. For brevity, we only show relevant details in these examples.

An example of how ActiveRecord (default ORM for Ruby on Rails) can be used to define a model is provided in Figure 2.1, lines 1-18. The example application defines four ActiveRecord classes: `User`, `Project`, `Todo` and `Note`, declared in lines 1-6, 7-11, 12-15 and 16-18 respectively.

Each class contains a set of associations (relations) with other classes. These associa-

```
1  class User
2    devise :database_authenticable
3    enum role: [:admin, :nonadmin]
4    has_many :todos
5    has_many :projects
6  end
7  class Project
8    belongs_to :user
9    has_many :todos
10   has_many :notes
11 end
12 class Todo
13   belongs_to :user
14   belongs_to :project
15 end
16 class Note
17   belongs_to :project
18 end
19
20 class Ability
21   def initialize(user)
22     can [:index, :show], :all
23     if user.admin?
24       can :manage, :all
25     else
26       can :manage, User, id: user.id
27       can :manage, Project, user_id: user.id
28       can :manage, Todo, user_id: user.id
29       can :manage, Note
30     end
31   end
32 end
```

```
34 class TodosController
35   def create
36     @project = Project.find(params[:project_id])
37     @user = current_user
38     @todo = Todo.new
39     @todo.user = @user
40     @todo.project = @project
41     @todo.save!
42     respond_to(...)
43   end
44 end
45 class ProjectsController
46   def destroy
47     @project = Project.find(params[:project_id])
48     raise unless can? :destroy, @project
49     @project.notes.each do |n|
50       n.delete
51     end
52     @project.delete
53     respond_to(...)
54   end
55 end
```

Figure 2.1: Excerpt from a Rails application

tions are declared using methods `belongs_to`, `has_one`, `has_many` and `has_and_belongs_to_many` that imply different cardinality and schema details. Figure 2.2 shows the class diagram corresponding to the code given in Figure 2.1. For example, each `Todo` object has at most one associated `Project` (line 14). The types and symmetry of associations are inferred from association names. For example, `Project.todos` and `Todo.project` are symmetrical: for every `Project` p and every `Todo` t of that `Project`, the `Project` of t is p.

Lines 20-32 define the Ability class. The Ability class is used to declare the access control policy using CanCan [20], a role-based access control library for Rails. The developer has to implement the Ability class in order to use CanCan. Every time an

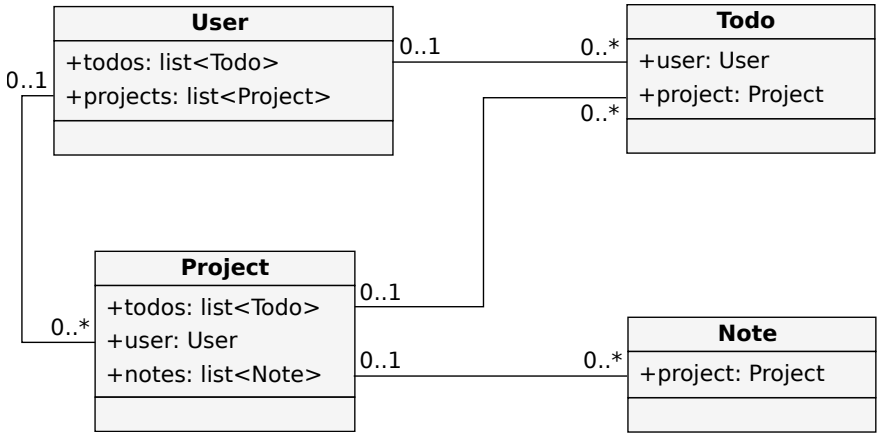


Figure 2.2: Class diagram corresponding to Figure 2.1

action is invoked, an ability object that corresponds to the user that made the request is created. Typically, the way this object is initialized depends on the role of the user making the request. Once this object has been created, it can be queried during the action's execution to check if certain operations are permissible for the current user.

The constructor of an Ability object takes a single argument, the User object representing the user who invoked the action, referred to as the current user (line 21). Inside the constructor we see branches and `can` statements. These statements whitelist certain operations on certain objects. For example, in line 22, the current user is permitted to execute operations `index` and `show` on objects of all types (special keyword `all`). These operations by convention correspond to action names. Then, if the user has the `admin` role (line 23), in line 24 he is permitted to execute all operations (special keyword `manage`) on all objects. Otherwise, if the user does not have the `admin` role (line 25), in line 26 he is permitted to manage all User objects whose `id` is equal to the `id` of the current user. In other words, a user can do all actions on their own user object. In lines 27 and 28 it is declared that a user can manage all Projects and Todos whose `user_id` match the `id` of the current user. In other words, each nonadmin can execute any action on all Projects and Todos that belong to them as defined by associations in lines 8 and 13. Finally, in

line 29, it is declared that a nonadmin can manage all Note objects.

Two actions can be seen in Figure 2.1: one in `TodosController` called `create` (lines 35-43) and one in `ProjectsController` called `destroy`(lines 46-54).

The `TodosController#create` action takes an argument as part of the request, called `project_id`. This argument is used to lookup the corresponding `Project` object and assign it to a variable `@project` (line 36). In line 37, the current user object is stored in a variable called `@user`. The action, then, creates a new `Todo` instance (line 38), associates it with the loaded user and project objects (lines 39 and 40) and saves the changes (line 41). The response is synthesized in line 42 by the view, which is omitted for brevity.

The `TodosController#destroy` action (lines 46-54) takes a single request argument `project_id`. The corresponding `Project` object is loaded in line 47 and stored in a variable. Line 48 serves to enforce access control. The `can?` method will check whether the Ability object that corresponds to the current user can execute the `:destroy` operation on the loaded project, as defined by the Ability object. An exception will be raised if that is not the case. If we look at the Ability class, there are two ways in which this operation can be allowed: either in line 24 if the current user is an admin, or in line 27 if the project in question is associated with the current user. Assuming the access control check passes, in line 49, the action iterates through all `Notes` associated with that project and deletes them one at a time (line 50). Finally, said project gets deleted (line 52).

Assume that we would like to verify the following property for the application in Figure 2.1: *Each* `Todo` *object is associated with a* `Project` *object.* In order to do that, we first need a way to express this property. We developed a Rails library for specification of data model invariants using Rails syntax. For example, this property would be stated as:

```
invariant forall{ |todo| not todo.project.empty? }
```

For this property, our tool would show that the `TodosController#create` action pre-

serves the given invariant, whereas the `ProjectsController#destroy` action potentially violates the invariant. If the deleted project had Todo objects associated with it at the beginning of the action, after deleting it, these Todo objects will be left with no associated Project, invalidating the invariant.

Similarly, we could use our tool to verify access control enforcement. For example, the `TodosController#create` action creates a `Todo` object without any enforcement of access control. However, if we take a closer look, we observe that this action does not violate the access control policy. Note that the only side-effect of this action is that it creates a `Todo` object. If the current user is an admin, this is permitted by the `can` statement in line 24. Otherwise, if the current user is not an admin, considering that the new `Todo`'s user is always the current user, line 28 permits this object creation.

2.2 Abstract Data Stores

In this section we define the model that we extract from a web application. The *abstract data store*, or *data store* in short, is an abstraction of a web application that focuses on the persistent data that the application manages.

An abstract data store (or just *data store*) is a structure $DS = \langle C, L, A, I, R, P \rangle$ where C is a set of classes, L is a set of associations, A is a set of actions, I is a set of invariants, R is a tuple defining user roles, and P is a set of permissions.

A *data store state* is a tuple $\langle O, T, U \rangle$ where O is the set of objects, T is the set of tuples denoting associations among objects, and U is the set of role assignments for users. We define $\overline{DS}$ to be the set of all data store states of DS.

2.2.1 Classes and Objects

The set of classes C identifies the types of objects that can be stored in the data store. Each class can have a set of superclasses (superclass(c) $\subset C$) and, transitively, the superclass relation cannot contain cycles. We will use operator $c_c < c_p$ to denote that c_p is a parent class to c_c, transitively or directly. We will use operators $>$, $\leq$ and $\geq$ accordingly.

One class $c_U \in C$ is called the *authenticable class*. The authenticable class is the class whose objects represent users of the application. Typically it is called just `User`. The concept of the user class is used in Rails whenever authorization and/or authentication take place, and it makes it straightforward to associate users with data that belongs to them.

For example, given the application presented in Figure 2.1, C would encompass four classes: `User`, `Project`, `Todo` and `Note`, and `User` is the authenticable class $c_U =$ `User`. The superclass set of each of these classes is empty.

Given a data store state $\langle O, T, U \rangle \in \overline{DS}$, O is the set of objects that are stored in a data store at some point in time. Each object $o \in O$ is an instance of a class $c \in C$ denoted by $c = \text{classof}(o)$. We use the notation O_c to encapsulate all objects in O whose class is c or any subclass of c. We define $\overline{O}$ to be the set of all sets of objects that appear in $\overline{DS}$.

In each data store state $\langle O, T, U \rangle \in \overline{DS}$, there exists a special object $o_U \in O_{c_U}$ that represents the *current user object*. The current user object represents the user who invoked the action.

2.2.2 Associations and Tuples

An association $l = \langle name, c_o, c_t, card \rangle \in L$ contains a unique identifier $name$, an origin class $c_o \in C$, a target class $c_t \in C$ and a cardinality constraint $card$. Cardinality constraints supported by ORM tools are limited, and so is our definition of valid cardinality constraints. Cardinality constraints are a pair of ranges n_o and n_t written as n_o-n_t. Ranges n_o and n_t describe the allowed number of objects on the origin and target side of the association respectfully. The possible ranges are: $[0 \dots 1]$, 1, $[1 \dots *]$ and $*$. For example, cardinality constraint $1 - *$ defines that every target object is associated with exactly one origin object. Alternatively, cardinality constraint $[0 \dots 1] - 1$ defines that every object of the target class is associated with an object of the origin class, and that no object of the origin class is associated with more than one object of the target class.

For example, given the application presented in Figure 2.1, there are four associations in L:

$$l_1 = \langle \texttt{User\_todos}, \texttt{User}, \texttt{Todo}, 1 - * \rangle$$

$$l_2 = \langle \texttt{User\_projects}, \texttt{User}, \texttt{Project}, 1 - * \rangle$$

$$l_3 = \langle \texttt{Project\_todos}, \texttt{Project}, \texttt{Todo}, 1 - * \rangle$$

$$l_4 = \langle \texttt{Project\_notes}, \texttt{Project}, \texttt{Note}, 1 - * \rangle$$

Similar to how objects are instances of classes, tuples are instances of associations. Each tuple $t \in T$ is in the form $t = \langle l, o_o, o_t \rangle$ where $l = \langle name, c_o, c_t, card \rangle \in L$ and $\text{classof}(o_o) \leq c_o$ and $\text{classof}(o_t) \leq c_t$. For a tuple $t = \langle r, o_o, o_t \rangle$ we refer to o_o as the origin object and o_t as the target object.

Note that we did not define that data store states need to have association cardinality correctly enforced. This is because, sometimes, an action will temporarily invalidate

cardinality while mutating data. In fact, cardinality is enforced when the data gets sent to the database, either by application-level validations or by the database schema directly. If cardinality constraints are violated, the action should abort without modifications to the data, trivially preserving all invariants. This behavior is not interesting for our purpose. Hence, we treat cardinality constraints as implicit invariants that are necessarily correct before and after an action executes.

2.2.3 Actions

Given a data store $DS = \langle C, L, A, I, R, P \rangle$, A denotes the set of actions. Actions are used to query or update the data store state. Each action $a \in A$ is a set of *executions* $\langle s, s', \alpha \rangle \subseteq \overline{DS} \times \overline{DS} \times \overline{O}$ where $s = \langle O, T, U \rangle$ is the pre-state of the execution, $s' = \langle O', T', U' \rangle$ is the post-state of the execution, and $\alpha \subseteq O'$ is the set of objects shown to the user as the result of this action's execution.

Given an action $a \in A$ and an execution $\langle s, s', \alpha \rangle \in a$, we can define the sets of objects this execution created, deleted, and read as follows:

$$o \in \mathit{created}(\langle s, s', \alpha \rangle) \Leftrightarrow o \notin s \wedge o \in s'$$

$$o \in \mathit{deleted}(\langle s, s', \alpha \rangle) \Leftrightarrow o \in s \wedge o \notin s'$$

$$o \in \mathit{read}(\langle s, s', \alpha \rangle) \Leftrightarrow o \in \alpha$$

In practice, an action is not an arbitrary set of state transitions. Instead, it is a sequence of statements. Statements are state transitions specified using a combination of boolean and object set expressions. Boolean expressions have the usual semantics, and object set expressions represent a set of objects of a common class. We further discuss the specifics of this language in Section 2.4. For now, the `TodosController#create` action corresponds

to the AST presented in Figure 2.3.

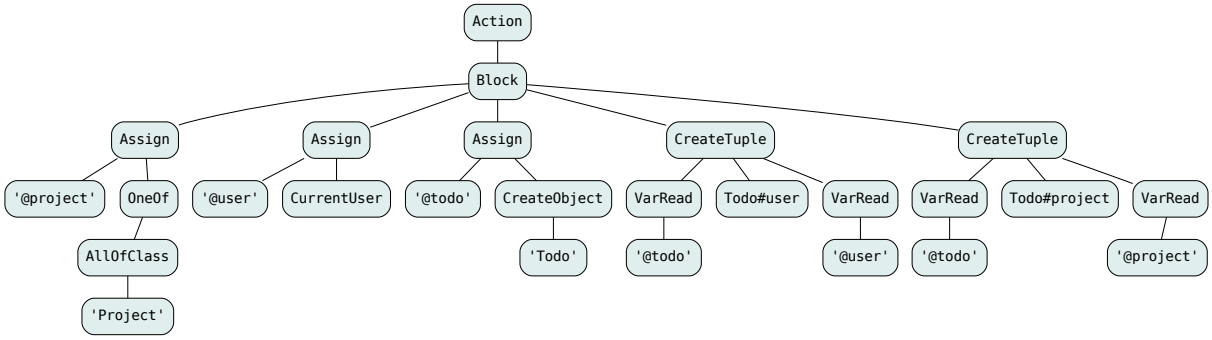


Figure 2.3: Action translation example

2.2.4 Invariants

Given a data store $DS = \langle C, L, A, I, R, P \rangle$, I is the set of invariants. An invariant $i \in I$ corresponds to a function $i \colon \overline{DS} \to \{\text{false}, \text{true}\}$ that identifies the set of data store states which satisfy the invariant.

2.2.5 Roles and Role Assignments

In a data store $DS = \langle C, L, A, I, R, P \rangle$, R is a tuple $\langle \overline{R}, R_x \rangle$ where $\overline{R}$ is the set of *roles*, and R_x is a *role constraint*.

Roles are used to distinguish different types of users. For example, a role could be defined to distinguish administrators from other users, or employees, etc.

In a data store state $\langle O, T, U \rangle$, $U \subseteq O_{c_U} \times \overline{R}$ is a set of role assignments. Each role assignment $\langle o, r \rangle \in U$ defines that user object o has user role r in this data store state.

The role constraint R_x is a function that maps a data store state to a boolean: $R_x \colon \overline{DS} \to \{true, false\}$. The rule constraint ensures a data store state's role assignments are semantically correct. This is application dependent and can be an arbitrary

condition, but in practice it often declares that each user has to have exactly one user role.

For example, line 3 of Figure 2.1 declares the set of roles $\overline{R}$ to be $\{\texttt{admin}, \texttt{nonadmin}\}$. In addition, it defines that roles `admin` and `nonadmin` are mutually exclusive:

$$R_x(\langle O, T, U\rangle) \Leftrightarrow (\forall o \in O_{c_u} \colon \langle o, \texttt{admin}\rangle \in U \Leftrightarrow \langle o, \texttt{nonadmin}\rangle \notin U)$$

2.2.6 Permits

Given a data store $DS = \langle C, L, A, I, R, P\rangle$, P is the set of permits. Permits are used to whitelist operations on a data store, depending on the role of the user executing the operation and the object the operation is executed on.

Each permit $p \in P$ is a tuple $\langle g, ops, e\rangle$ where $g \subseteq \overline{R}$ is a non-empty set of roles to which the permit applies, $ops \subseteq \{create, delete, read\}$ is a non-empty set of operations permitted by this permit, and e is an expression $e \colon \overline{DS} \to \overline{O}$ that maps a data store state to a set of objects: $e(\langle O, T, U\rangle) = \alpha$ such that $\alpha \subseteq O$. The expression e is used to determine the set of objects to which the permit applies. Note that, because we abstract basic type fields away in our model, updating an object is outside our domain of abstraction. Updating associations is equivalent to atomically deleting and creating tuples. As such, we can only consider these three operations.

To demonstrate how permits correspond to web applications, line 22 of Figure 2.1 defines that all `admin`s and `nonadmin`s are permitted to *read* all objects. This corresponds to the following permit:

$$p = \langle \{\texttt{admin}, \texttt{nonadmin}\}, \{read\}, e\rangle, \text{ where } o \in e(\langle O, T, U\rangle) \Leftrightarrow o \in O$$

Similarly, line 26 of Figure 2.1 declares that all `nonadmin`s can do any operation on all

`Projects` that belong to the current user. Let l be the "User.projects" association, and o_U the current user object. This `can` declaration corresponds to the following permit:

$$p = \langle \{\texttt{nonadmin}\}, \{create, read, delete\}, e \rangle, \text{ where } o \in e(\langle O, T, U \rangle) \Leftrightarrow \langle l, o_U, o \rangle \in T$$

In words, this is a permit $\langle \{\texttt{nonadmin}\}, \{create, read, delete\}, e \rangle$ where e is defined as an expression that, when given a state $\langle O, T, U \rangle$, contains all objects o that are associated to the current user object via association l.

2.2.7 Behaviors

A *behavior* of a data store is an infinite sequence of data store states such that the initial state satisfies all invariants, and each pair of consecutive states is covered by at least one action. Formally, given a data store $DS = \langle C, L, A, I, R, P \rangle$, a behavior of a data store DS is an infinite sequence of data store states $\langle O_0, T_0, U_0 \rangle$, $\langle O_1, T_1, U_1 \rangle$, $\langle O_2, T_2, U_2 \rangle, \ldots$ where

- For all $k \geq 0$, $\langle O_k, T_k, U_k \rangle \in \overline{DS}$ and there exists an action $a \in A$ such that $(\langle O_k, T_k, U_k, \rangle, \langle O_{k+1}, T_{k+1}, U_{k+1} \rangle, \alpha) \in a$ for some $\alpha \subset O_{k+1}$, and
- $\forall i \in I \colon i(\langle O_0, T_0, U_0 \rangle) = \text{true}$

Given a data store $DS = \langle C, L, A, I, R, P \rangle$, all states that appear in a behavior of DS are called *reachable states* of DS and denoted as $\overline{DS}_R$.

2.3 Data Store Correctness

We use our approach to verify correctness of abstract data stores in two respects: data integrity and access control.

2.3.1 Data Integrity

Given an abstract data store $DS = \langle C, L, A, I, R, P \rangle$, we call DS *consistent* if and only if all reachable states of DS satisfy all the invariants of DS, i.e., DS is consistent if and only if for all $\langle O, T, U \rangle \in \overline{DS}_R$, for all $i \in I$, $i(\langle O, T, U \rangle) = \text{true}$. The verification problem for data integrity is to determine if a given abstract data store is consistent. Since we do not bound the sizes of the classes and relations in a data model, and since we allow arbitrary quantification in invariant properties, determining if a data store specified in the ADS language is consistent or not is an undecidable verification problem.

As we discussed earlier, in RESTful applications, each action is required to preserve the invariants of the data model independently of the previous execution history. This is a stronger requirement that implies the consistency condition defined above, and can be formulated as inductive invariant verification. An inductive invariant is a property where given a state that satisfies the property, all the next states of that state also satisfy the property. In other words, an inductive invariant is a property that is preserved by all transitions (i.e., all actions) of a given system. An abstract data store $DS = \langle C, L, A, I, R, P \rangle$ is consistent if the conjunction of all the invariants $i \in I$ is an inductive invariant. In other words, an abstract data store $DS = \langle C, L, A, I, R, P \rangle$ is consistent if and only if every execution of every action preserves all invariants:

$$F_{cons} \equiv \forall a \in A \colon \forall \langle s, s', \alpha \rangle \in a \colon (\forall i \in I \colon i(s)) \Rightarrow (\forall i \in I \colon i(s'))$$

2.3.2 Access Control

Authorization is fundamentally about ensuring that users can view and modify only data that they have been permitted to view and modify, using a set of methods that are permitted to them. The goal of approach is to check whether all operations that could

be executed by any action and for any user are permissible with respect to the access control policy.

Given a data store $DS = \langle C, L, A, I, R, P \rangle$, a permit $p = \langle g, ops, e \rangle \in P$ *accepts* an operation op on an object o in state s, denoted as $p[op, s, o]$, if and only if the current user o_U has at least one role from g, the operation op is in ops, and o is inside the set that e evaluates to in data store state s. Formally:

$$p[op, s, o] \Leftrightarrow (\exists r \in g \colon \langle o_U, r \rangle \in U) \wedge op \in ops \wedge o \in e(s)$$

Now that we defined how to check permissions for a given operation in a given state for a given set of objects, we need to extend this check to cover all possible behaviors.

One question we need to answer before we can do that is: In which state of an action's execution should we check for permissions? If we choose the pre-state, it becomes impossible to check permissions for object creation because the created objects do not yet exist in the pre-state, as well as tuples that might be necessary to check the access control policy correctly. Similarly, it is impossible to evaluate permissions for the delete operation in the post-state of an execution. In order to handle all possible scenarios, we chose to evaluate creation permissions in the post-state (once all the objects and tuples have been created), deletion permissions in the pre-state (before any objects or tuples have been deleted), and read permissions in the post-state (as this is the state shown to the user of the application).

We define whether an action $a \in A$ of a data store $DS = \langle C, L, A, I, R, P \rangle$ correctly enforces the access control policy as follows. An action correctly enforces the access control policy if and only if, for every execution in a:

- There exists a permit that accepts the creation of every object created by this execution,

- There exists a permit that accepts the deletion of every object deleted by this execution, and
- There exists a permit that accepts the read operation on every object read by this execution.

Formally, an action a correctly enforces the access control policy if and only if:

$$\begin{aligned}
&\forall \langle s = \langle O, T, U \rangle, s' = \langle O', T', U' \rangle, \alpha \rangle \in a\colon \\
&\quad \forall o \in \mathit{created}(\langle s, s', \alpha \rangle)\ \exists p \in P\colon p[\{\mathit{create}\}, s', o] \\
&\quad \wedge \forall o \in \mathit{deleted}(\langle s, s', \alpha \rangle)\ \exists p \in P\colon p[\{\mathit{delete}\}, s, o] \\
&\quad \wedge \forall o \in \mathit{read}(\langle s, s', \alpha \rangle)\ \exists p \in P\colon p[\{\mathit{read}\}, s', o]
\end{aligned}$$

For example, let us take a look at the `TodosController#create` action in Figure 2.1. This action will, in its every execution $\langle s, s', \alpha \rangle$, create a single Todo object as well as a pair of tuples. No matter the role of the current user, this creation is covered by the access control policy: if the current user is an `admin`, by a permit that corresponds to line 24. Otherwise, if the current user is a `nonadmin`, this creation is covered by the permit that corresponds to line 27 because, in state s', the newly created object will be associated to the current user.

Let us examine the `ProjectsController#destroy` action in Figure 2.1. Logically, if any object is deleted, then the action has not been aborted in line 48. Therefore, the current user is either an `admin` (in which case the permit in line 24 accepts the object and the operation), or the current user is a `nonadmin` but is also the user of the project (which is covered by the permit in line 27). Since there exists no execution in which the project is deleted without proper authorization, this action correctly enforces the access control policy with regards to deleting objects of the Project class.

2.4 Abstract Data Store Language

Abstract Data Store Language (ADSL) is a language for describing ADSs. Since we extract ADS language specifications from existing applications, ADSL is an intermediate language whose specifications are represented by abstract syntax trees. The abstract syntax tree for an ADS specification contains a set of `Class`, `Association`, `Action`, `Invariant`, `Role` and `Permit` nodes corresponding to the model $DS = \langle C, L, A, I, R, P \rangle$.

Following the formal definition of C and L given in Section 2.2, a `Class` node may refer to other `Class` nodes as superclasses, and contains any number of `Association`s. `Association` nodes are defined by name, target `Class` and cardinality. Modifiers exist to denote mutually symmetrical associations.

The most complex part of an ADS specification are action specifications. `Action` nodes typically contain a `Block` node, which in turn contains any number of other statement nodes. Each statement node may migrate the data store state, or be evaluated to a boolean or set of objects, or both. Statement nodes that return an object set will return a set of objects in the data store that share a common class or superclass. Statement nodes that return a boolean will evaluate to *true* or $false$, typically used as conditions in branches. Some statement nodes also use variable, class or association names as arguments.

For example, the `AllOf(class)` node returns all objects of class (or subclass of) `class`. Most statement nodes rely on other statement nodes to fully define their behavior: for example, the `Subset(e)` node will have the same side-effects as e, and return a subset of the object set e returned.

As another example, the `OneOf(object set)` node represents a non-deterministic selection of one object from its argument object set `object set`. This node also implicitly defines that the argument will have at least one object inside it. The `TryOneOf(object`

Node	Children	State Migrations	Returns
`Block`	**Stmt*	State migrations of the arguments applied sequentially	The last child's return value, or an empty object set if there are no children
`CreateObject`	Class name	Creates a new object of the stated class that is not associated to any other object	The singleton set containing this object
`Delete`	*Object Set*	Deletes objects belonging to the object set, as well as all tuples tied to these objects	Empty object set
`CreateTuple`	*Object Set*, Association, *Object Set*	Associates all objects from the two object sets over the association	The second object set
`DeleteTuple`	*Object Set*, Association, *Object Set*	Disassociates all objects from the two object sets over the association	The second object set
`If`	*Boolean*, *Node*, *Node*	Migrations of the condition, followed by only the migrations of the appropriate branch node	Return value of the branch that corresponds to the result of the branch condition
`ForEach`	Variable name, *Object Set*, *Block*	Executes the block once for each object in the given object set, assigning the singleton set of this object to the variable for each iteration	Empty object set
`Assign`	Variable name, *Object Set*	Assigns the object set returned by the object set to the variable	The object set assigned to the variable
`DereferenceCreate`	*Object Set*, Association	Creates a new object and associates it with all objects from the supplied object set over the association	The singleton set containing the newly created object
`Raise`	-	Defines the program path to be unreachable	-

Table 2.1: State-migrating abstract data store statement nodes

`set)` statement node does not implicitly define the argument to be non-empty: if the argument is empty, it will return an empty set itself. Otherwise it will return a singleton subset of the argument.

The list of all statement nodes in given in Tables 2.1 and 2.2. This list is presented in two tables for space reasons, as we allow nodes from both lists to be used as arguments for either list. Table 2.1 lists nodes with side effects: these nodes migrate the state when they are evaluated. They may or may not return a value when evaluated. Table 2.2 lists nodes that have no side effects. In both tables we use the words *Object Set* and *Boolean* to imply strict type requirements. Furthermore, in both tables, we omit explicitly listing state migrations of the children as part of the state migrations of a node. In all cases, state migrations of children are applied before any state migrations of a node in question, in listed order.

`Invariant` nodes are represented using boolean nodes as defined in Table 2.2. Note

Node	Children	Returns
`VarRead`	Variable name	The set of objects assigned to the variable that were not deleted since the assignment
`AllOf`	Class name	Contains all objects of the given class, including subclasses
`Subset`	*Object Set*	Contains a subset of the argument object set
`TryOneOf`	*Object Set*	Exactly one object from the given object set, or empty if the given object set is empty
`OneOf`	*Object Set*	Exactly one object from the given object set, enforcing the argument object set to be non-empty
`Union`	**Object Set*	Union of given object sets. Legal only if there exist a common type between said object sets
`Empty`	-	Contains no objects
`Dereference`	*Object Set*, Association	All objects associated with at least one object from the given object set over the association
`And`, `Or`, `Implies` etc.	**Boolean*	Expected semantics, evaluating all expressions in the order in which they are listed
`ForAll`, `Exists`	`Variable`, *Object Set*, *Boolean*	Expected quantification semantics over object sets
`=`, $\subset$ `etc.`	**Object Set*	Expected set operators
`IsEmpty`	*Object Set*	Evaluates to *true* iff supplied *Object Set* is empty
`True`, `False`	-	Boolean constants
`*`	-	Non-deterministic boolean value
`CurrentUser`	-	A singleton set containing the current user object
`InUserGroup`	*Object Set*, Role name	Returns *true* iff all objects in the object set have the argument role. Legal only if it is an object set of the authenticable class

Table 2.2: State-preserving abstract data store statement nodes

that invariants are forbidden from migrating state.

`Role` nodes are used to declare user roles. They contain no information other than the name of the role. For example, Figure 2.1 defines two role nodes: one for a role named `admin`, and one for `nonadmin`. The `Role constraint` is a single boolean statement node that is defined to always hold. Like invariants, role constraints cannot migrate state.

Finally, `Permit` nodes have a non-empty set of role nodes as children, a non-empty set of operations (chosen among `read`, `create` and `delete`), and an object set node that represents its expression. This object set node cannot migrate state. Corresponding to

the example in Figure 2.1 are the following permit nodes:

$$p_1 = \langle \{\texttt{admin}, \texttt{nonadmin}\}, \{read\}, \text{AllOf}(\texttt{User}) \rangle$$

$$p_2 = \langle \{\texttt{admin}, \texttt{nonadmin}\}, \{read\}, \text{AllOf}(\texttt{Project}) \rangle$$

$$p_3 = \langle \{\texttt{admin}, \texttt{nonadmin}\}, \{read\}, \text{AllOf}(\texttt{Todo}) \rangle$$

$$p_4 = \langle \{\texttt{admin}, \texttt{nonadmin}\}, \{read\}, \text{AllOf}(\texttt{Note}) \rangle$$

$$p_5 = \langle \{\texttt{admin}\}, \{read, create, delete\}, \text{AllOf}(\texttt{User}) \rangle$$

$$p_6 = \langle \{\texttt{admin}\}, \{read, create, delete\}, \text{AllOf}(\texttt{Project}) \rangle$$

$$p_7 = \langle \{\texttt{admin}\}, \{read, create, delete\}, \text{AllOf}(\texttt{Todo}) \rangle$$

$$p_8 = \langle \{\texttt{admin}\}, \{read, create, delete\}, \text{AllOf}(\texttt{Note}) \rangle$$

$$p_9 = \langle \{\texttt{nonadmin}\}, \{read, create, delete\}, \text{CurrentUser} \rangle$$

$$p_{10} = \langle \{\texttt{nonadmin}\}, \{read, create, delete\}, \text{Dereference}(\text{CurrentUser}, \text{projects}) \rangle$$

$$p_{11} = \langle \{\texttt{nonadmin}\}, \{read, create, delete\}, \text{Dereference}(\text{CurrentUser}, \text{todos}) \rangle$$

$$p_{12} = \langle \{\texttt{nonadmin}\}, \{read, create, delete\}, \text{AllOf}(\texttt{Note}) \rangle$$

Chapter 3

Model Extraction

In this chapter we present *symbolic model extraction*, an approach for extracting models from programs written in dynamically typed programming languages.

The key ideas of symbolic model extraction are 1) to use the source language interpreter for model extraction, which enables us to handle dynamic features of the language, 2) to use code instrumentation so that execution of each instrumented piece of code returns the formal model that corresponds to that piece of code, 3) to instrument the code dynamically so that the models of methods that are created at runtime can also be extracted, and 4) to execute both sides of branches during instrumented execution so that all program behaviors can be covered in a single instrumented execution.

We theoretically generalize the approach to different languages and models, and implement evaluate it for extracting abstract data models from Rails applications.

Section 3.1 demonstrates why model extraction can be difficult given the dynamic features of Ruby, by discussing a small portion of a Rails application. Section 3.2 defines our approach. Section 3.3 applies the approach for extraction of abstract data models from Rails applications, and Section 3.4 experimentally evaluates our extraction method.

```
1  class Article < ActiveRecord::Base
2    acts_as_paranoid
3  end
4  class ArticlesController < ApplicationController
5    load_resource
6    before_action :destroy do
7      redirect_to :back unless current_user.verified?
8    end
9    def destroy
10     @article.destroy!
11   end
12 end
```

Figure 3.1: Example of Rails dynamic features

3.1 Model Extraction and Dynamic Features of Ruby

Consider the excerpt of a Rails application in Figure 3.1. Lines 1-3 declare a model class called `Article`. This particular class defines articles that are managed by this web application. This class does not contain any fields or additional methods for the sake of brevity. Lines 4-12 define the `ArticlesController`, which contains one action called `destroy` (lines 9-11). This action seemingly deletes the object stored in the `@article` variable by invoking the `destroy!` method on it in line 10.

Ruby is a dynamically typed language and lets the developer freely define and replace existing methods at runtime. In this example, the dynamic features of Ruby are used to such an extent that the action's source code deceptive. The action does much more than deleting an object.

First, it is not clear which article object is being deleted. In line 5 of Figure 3.1 we see the `load_resource` declaration, defined by the CanCan gem [20]. This declaration will ensure that, before an action executes, the framework will preload an object and store it in a variable, to be accessed from inside the action. The specifics of this preload operation are subject to a number of conventions such as the name of the controller, the name of the action, and configurations.

Second, the `before_action` declaration in lines 6-8 prepends a *filter* to the action.

```
1  class ArticlesController < ApplicationController
2    def destroy
3      @article = Article.find(params[:id])
4      redirect_to :back unless current_user.verified?
5      @article.deleted_at = Time.now
6      @article.save!
7    end
8  end
```

Figure 3.2: Static equivalent to action in Figure 3.1.

Filters execute before or after an action and are usually used to prepare data for an action, or to conditionally prevent an action from executing any further. In this case, if the current user is not verified (line 7), the filter will redirect to a different page. This redirection will prevent the action from executing.

Finally, in line 10, the action invokes the `destroy!` method on the object in order to delete it. However, in line 2, the `acts_as_paranoid` declaration (provided by the ActsAsParanoid gem [84]), overrides the `destroy!` method for the Article class. Instead of deleting an object, the object is simply marked as deleted but not removed from the database. This allows for Article objects to be restored later if need be.

Figure 3.2 contains a destroy action that is semantically equivalent to the action in Figure 3.1, but with its semantics transparent and directly understandable from source code.

This is a simple example of how actions can be enhanced using dynamic features of the Ruby language. There exists a rich set of libraries that Rails developers can use to enhance the framework. Some of these libraries, such as ActiveAdmin [3], can even generate entire actions that are not present statically.

Dynamic method generation is not limited to advanced Rails features. Even core Ruby use dynamic method generation to implement basic functionality. For example, to declare fields that objects of a Ruby class have, one might use code:

```
class Class
  attr_accessible :field1, :field2
end
```

The `attr_accessible` declaration is actually a class-level method call that dynamically generates getters and setters for all listed fields. Similarly, association declarations (lines 4-5, 8-10, 13-14 and 17 in Figure 2.1 are core Rails methods that dynamically generate other methods based on the state of the class and the arguments. As such, every Rails application uses dynamic method generation.

In our experience, in practice, method generation in Rails is *input independent*: inputs coming from the user do not dictate the kind of methods that are generated. For example, the `attr_accessible` declaration will generate getters and setters at class load time, independently from any actions that may be invoked by the user. Even methods that are generated during action execution are input independent: the same methods would be generated for every action execution regardless of user inputs. As such, by executing the application without regard to user input, we are able to capture dynamically generated methods and extract their models.

Dynamic method generation in applications makes static analysis and model extraction difficult, as the semantics of an application are fully defined only at runtime, after libraries have had the opportunity to augment them. This problem has been observed [56], and the solution typically involves manual modeling of source code or semantics of dynamically generated methods. This process, while helpful, is prone to error as the manually written model may not match the actual semantics of the program. This is especially dangerous in case a third-party library overrides the expected semantics of a core Rails method. Our approach, conversely, will capture the correct semantics of the program without manual effort and with a higher level of confidence that the extracted model correctly matches the application.

We explain symbolic model extraction and its application to data model verification in Section 3.3, but to demonstrate the purpose of symbolic extraction, it extracts the ADS

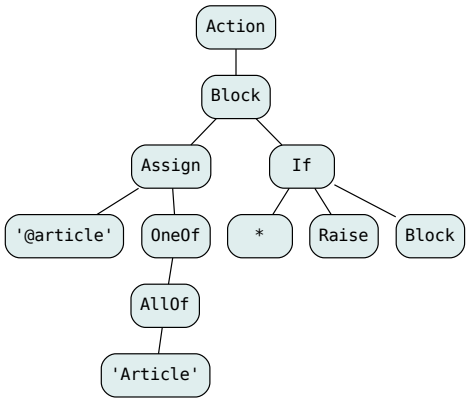


Figure 3.3: Model extracted from Figure 3.1.

model in Figure 3.3 from the Rails program in Figure 3.1. This model is an abstraction of the original method. The `OneOf` node defines that an Article object is read from the database, without specifics on which object is loaded, and the `Assign` node stores this object in a variable. The `If` node will, because of a non-deterministic condition (node `*`), either `Raise` an exception and abort the action or execute an empty `Block` (equivalent to a noop). Finally, the model will correctly omit the delete operation that was seemingly present in the original source code.

3.2 Symbolic Model Extraction

We explain symbolic model extraction on an abstract programming language L that captures the core features of the languages such as JavaScript, Python or Ruby. Let us assume that L is an interpreted, dynamically typed, imperative programming language with functions as first-class citizens (e.g. functions can be assigned to variables, passed as arguments to function calls etc.).

For simplicity, we will represent a program written in L as a statement s. Since a sequence of statements is itself a statement, this perspective is accurate. At runtime,

programs written in L are executed using the interpreter I where I evaluates statements to migrate the program from one state to another state.

Because this is a dynamically typed language, the types of objects assigned to variables may change over time. In addition, the type system in the program can change in any number of ways: classes can be defined at runtime, methods can be added or even replaced at runtime.

Let $\overline{L}$ be the set of program states in L. These states include the program counter, the stack and heap memory states. This lets us define a statement s as a set of state transitions:

$$s \subseteq \overline{L} \times \overline{L}$$

In words, given an initial state $l \in \overline{L}$, executing a statement s will migrate the program state to some state l' such that $\langle l, l' \rangle \in s$. Furthermore, we constrain the definition of statements to have at least one state transition from any program state[1]. Let S be the set of all statements in language L.

For model-based verification, in order to verify a program (statement) $s \in S$, we need to extract $s^\sharp$: the model of a statement s in some modeling language $L^\sharp$. This model is an abstraction of the original statement, meaning that if there are undesirable behaviors in s, they can be detected in $s^\sharp$.

Let $\overline{L^\sharp}$ be the set of abstract program states. Each abstract program state $l^\sharp \in \overline{L^\sharp}$ is a set of concrete program states:

$$l^\sharp \subseteq \overline{L}$$

[1]In practice, these languages raise exceptions if the statement is not normally executable from a program state. This behavior itself constitutes a state transition.

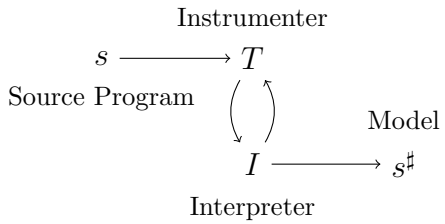


Figure 3.4: Overview of symbolic model extraction

Similarly, an abstract statement $s^\sharp$ is a set of transitions between abstract states that abstracts a concrete statement s. More precisely, for every state transition in s, $s^\sharp$ contains the transition of corresponding abstract states:

$$\forall \langle l, l' \rangle \in s \colon \exists \langle l^\sharp, l^{\sharp\prime} \rangle \in s^\sharp \colon l \in l^\sharp \wedge l' \in l^{\sharp\prime}$$

We can see that $s^\sharp$ simulates the behavior of s, i.e., for each behavior in s there exists a corresponding behavior in $s^\sharp$. Hence, $s^\sharp$ is an abstraction of s.

Notice that, in order to simplify the presentation, this definition of statements does not account for expressions and expression values. Where relevant, we use notation $I(s)$ to refer to the return value of an execution of s in some program state by the interpreter I.

3.2.1 Symbolic Model Extraction Rules

Symbolic model extraction uses the interpreter for the source language, and an instrumentation function that is accessible during runtime, to execute the input program in an instrumented, path-insensitive environment to explore static as well as dynamically generated code and extract the model for the given program in the target modeling language.

We illustrate the high level information flow in the symbolic model extraction in Fig-

ure 3.4 where the input program (s) in the source language is passed to the instrumenter function (T). The instrumenter will instrument the given program and pass it to the interpreter for execution. When new code is encountered or dynamically generated, the interpreter will pass this new code to the instrumenter for immediate instrumentation. The execution of the instrumented program returns the extracted model ($s^\sharp$) in the target modeling language.

In order for this approach to work without developing a custom interpreter, the instrumenter has to be implemented in the source programming language itself. Newly generated code can then be investigated and instrumented using metaprogramming.

Key to our approach is the *symbolic model extraction instrumentation function* T, or the *instrumenter* in short. T is a function $T\colon S \to S$ that, given a statement $s \in S$, returns the instrumented statement $T(s)$. When executed by the interpreter I, $T(s)$ evaluates to the model of s:

$$I(T(s)) = s^\sharp$$

In words, the instrumenter transforms a statement such that executing the transformed statement using the source language interpreter returns the model of the original statement.

After implementing T in the source language, we can use it to instrument and extract a model of a dynamic program using the source language interpreter. We surround the program's entry point with a call to the instrumenter, and the instrumenter will propagate instrumentation as new code is encountered. As instrumentation prepares instrumentation of all encountered code (that is not already instrumented), this ensures that all executed code is instrumented, with the obvious exception of the instrumenter itself.

Rule #	s	$T(s)$	$I(T(s))$
1	$\alpha_1; \alpha_2; \ldots; \alpha_n$	`ins_block(`$T(\alpha_1)$`, `$T(\alpha_2)$`, ..., `$T(\alpha_3)$`)`	$\alpha_1^\sharp; \alpha_2^\sharp; \ldots; \alpha_n^\sharp$
2	`fn(`$\alpha_1 \ldots \alpha_n$`)`	T`(fn)(`$T(\alpha_1) \ldots T(\alpha_n)$`)`	$fn^\sharp(\alpha_1^\sharp \ldots \alpha_n^\sharp)$
3	`if` α `then` β `else` γ	`ins_if(`$T(\alpha)$`, `$T(\beta)$`, `$T(\gamma)$`)`	`if`$^\sharp$`(`$\alpha^\sharp$`, `$\beta^\sharp$`, `$\gamma^\sharp$`)`
4	1 `while` α 2 β	`ins_while(`$T(\alpha)$`, `$T(\beta)$`)`	`while`$^\sharp$`(`$\alpha^\sharp$`, `$\beta^\sharp$`)`
5	$\alpha \, op \, \beta$	`ins_op(`op`, `$T(\alpha)$`, `$T(\beta)$`)`	$\alpha^\sharp \; op^\sharp \; \beta^\sharp$ if $op^\sharp$ is within $L^\sharp$ $*$ otherwise

6	`var =` α	`result =` $T(\alpha)$
		`var = SymVar.new(result.sym_type, 'var')`
		`ins_asgn('var', result)`
			`var =`$^\sharp$ $\alpha^\sharp$

7	`var`	`var`	`var`$^\sharp$

Table 3.1: Instrumentation for symbolic extraction

Implementing the instrumenter with regards to operations that are directly tied to the abstraction is generally straightforward. For example, if we are extracting a model of integer operations, T replaces integer addition so that, instead of returning the sum of two integers, integer addition returns a model of the addition operation.

The instrumenter's behavior is not obvious when it comes to dynamic language features, control flow, and data flow features such as scoping and assignments that appear in the source program. Table 3.1 demonstrates how the instrumenter could be implemented with regards to basic language constructs.

Sequences of statements

Rule 1 in Table 3.1 demonstrates how a sequence of statements is instrumented in order to extract the model of the sequence.

Let us extract the model of a sequence of statements $s = \alpha_1; \alpha_2; \ldots; \alpha_n$ using symbolic model extraction. Our goal is to instrument the sequence in such a way that executing the instrumented sequence returns the model of the sequence. To achieve this, we replace each statement α_k with its instrumented version $T(\alpha_k)$ and pass these instrumented

statements as arguments to an `ins_block` function. As such,

$$T(s) = \texttt{ins\_block}(T(\alpha_1), T(\alpha_2), \ldots, T(\alpha_n))$$

Note that this instrumented statement is still a valid statement in the source language. `ins_block` is method provided by our instrumentation library that merges a sequence of models of statements into a block model.

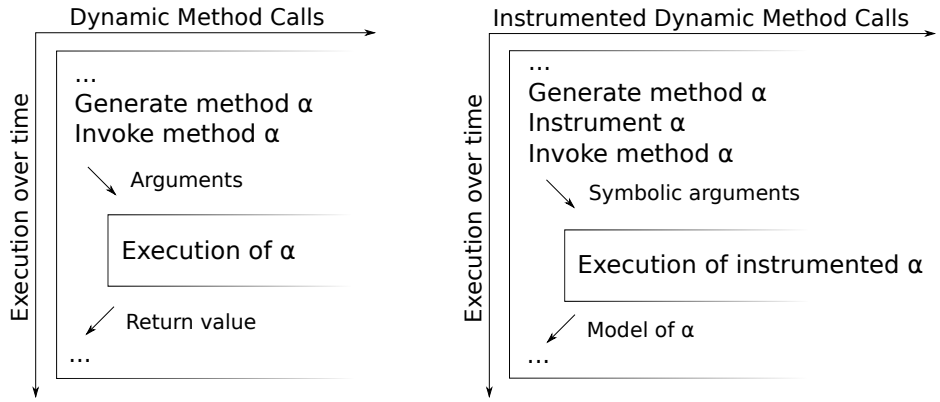
When executing $T(s)$, the interpreter will first evaluate each argument for `ins_block` in order. The result of each argument $T(\alpha_k)$ will be $\alpha_k^\sharp$, the model of the statement α_k. Finally, these models will be merged by `ins_block` into the sequence of statements in the modeling language.

Method calls

Rule 2 in Table 3.1 refers to how function (or method) calls are treated by the instrumenter. Any call to a function $fn(\alpha_1, \ldots, \alpha_n)$ in the source program is replaced with a call to $T(fn)(T(\alpha_1), \ldots, T(\alpha_n))$. In words, the instrumented function gets executed instead of the original function, with arguments having been instrumented as well. The result of this execution, as defined by the instrumenter, will be the model of the function's body.

To illustrate this approach, consider Figure 3.5(a). It illustrates the execution of a program that dynamically generates a method α and subsequently invokes it. Since code generation is done at runtime, it poses a problem for standard model extraction techniques. Figure 3.5(b) demonstrates how symbolic model extraction extracts the model of a dynamically generated method.

After generating the method, the interpreter instruments the generated method at runtime. The instrumentation alters the method such that, after it is invoked with



(a) Normal execution (b) Instrumented execution

Figure 3.5: Symbolic model extraction from dynamically generated methods.

symbolic arguments, the instrumented method returns its own model. This is how instrumentation is propagated through the program: all functions are instrumented just before they are invoked, allowing us to extract statically available source code as well as source code that might not exist statically.

Control Flow

Dynamic program analysis is typically subject to the problem of path explosion. Symbolic model extraction bypasses this problem by exploring all paths of the program at the same time. In order to achieve this, in order to extract the model of a branch, we execute both paths in an instrumented environment. After executing the branch in this manner in order to extract the models of both branches, we can continue to extract the model of subsequent statements as usual, again by executing them only once.

Rule 3 in Table 3.1 summarizes our approach for extracting models of branches. Given a branch where α is the condition and β and γ are the then and else block respectfully, the instrumenter replaces the branch with a call to `ins_if`$(T(\alpha)$, $T(\beta)$, $T(\gamma))$ which will consecutively instrument and execute the condition and both paths. The results of executions of instrumented elements are the models of each element, which are combined

into a model representation of the branch itself. Note that, contrary to intuition, this approach can handle some often encountered situations where different paths have seemingly conflicting side effects, such as assigning different values to the same variable. This will be made clear when discussing assignments further below.

Loops are handled analogously (Rule 4 in Table 3.1). Instead of executing the loop body a number of times, the loop can be instrumented and executed only once to extract the model of the loop.

Expressions

Rule 5 in Table 3.1 explains how our approach handles expressions. Even though this discussion assumes that the expression is a binary operator, the principle generalizes to any number of arguments.

Given an expression α *op* β, we first extract the model of α and β by instrumenting and evaluating them. Then, depending on the operation *op* itself and $\alpha^\sharp$ and $\beta^\sharp$, using the `ins_op` function, we return either the model that correctly abstracts the expression $(\alpha^\sharp op^\sharp \beta^\sharp)$, or $*$, representing any possible value.

The specifics of handling expressions depend on the source programming language, the target modeling language, and the desired abstraction. For example, if the abstraction handles integer addition and string concatenation, `ins_op` would check whether $\alpha^\sharp$ and $\beta^\sharp$ are both integers or both strings and return the model of the corresponding operation. If the types of arguments do not match, the result of `ins_op` would be a symbol representing any possible value.

Variables and scoping

Rule 6 in Table 3.1 refers to how variable assignments are treated by the instrumenter and Rule 7 explains how variable reads are treated by the instrumenter. These two

operations are closely tied to each other, as we actually instrument variable reads as a side effect of instrumenting assignments.

Given an assignment $var = \alpha$, the instrumenter should generate code that, when executed, returns the model representation of an assignment operation. Similarly, when a variable *var* is read in the original program, the corresponding model should represent the variable reading operation.

As shown in Rule 6 in Table 3.1, the instrumenter replaces the assignment with a sequence of three statements. The first statement instruments and extracts the model of the assigned expression, storing it in a temporary, local variable `result`.

The second statement creates a model of the variable read operation and stores it in the assigned variable, along with the type of the assigned expression and the name of the variable. That way, whenever this variable is subsequently read by the interpreter, the interpreter will identify the correct variable using the scoping rules of the language and return the proper variable read model. This not only reduces the amount of work needed to implement symbolic model extraction as we need not worry about variable scoping rules in the source language. Finally, the third statement constructs and returns the model of the assignment operation itself.

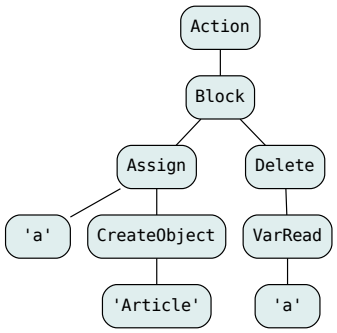
Since variables do not contain a value that is tied to the expression that was assigned to the variable, symbolic model extraction does not have a problem with the source program assigning different values to the same variable in different program paths. During symbolic model extraction from such a program, although a different models will be extracted from each assignment, all assignments will assign the very same value to the variable in question.

```
1  do
2    a = Article.new
3    a.destroy!
4  end
```

(a) Rails code

```
1  ins_block(
2    (
3      result = ins_call(Article, :new);
4      a = SymVar.new(result.sym_type, 'a');
5      ins_assignment(:a, result)
6    ),
7    ins_call(a, :destroy!)
8  )
```

(b) Instrumented Rails code



(c) Extracted model

Figure 3.6: Symbolic model extraction example.

Dynamic Features

Symbolic model extraction is built on the assumption that the dynamic features used in a program are input independent. Based on this assumption, during instrumented execution, methods are generated the same way they would be during normal execution, with symbolic values instantiated into concrete values of the appropriate type. This enables our symbolic model extraction technique to capture dynamically generated methods.

3.2.2 Symbolic Extraction Example

We will proceed to demonstrate how symbolic model extraction can be used for model extraction on an example. This example is designed to demonstrate key features of the technique, and how the approach deals with difficulties more straightforward techniques could not handle easily.

Assume that we wish to extract a model from the Ruby block in Figure 3.6(a). This block creates a new Article object (`Article.new`) and assigns it to a variable called `a` in line 2. In line 3 the `destroy!` method is invoked on the previously created object, deleting the object from the database. These statements are wrapped in a block (lines 1-4). The model we will eventually extract is presented in Figure 3.6(c)).

The instrumenter will automatically transform the block in Figure 3.6(a) to the block in Figure 3.6(b). This instrumented code follows the instrumentation rules previously discussed in Table 3.1.

The block presented in lines 1-4 of Figure 3.6(a) corresponds to the `ins_block` statement that spans lines 1-8 of Figure 3.6(b). This is a direct application of Rule 1 in Table 3.1. The two arguments of `ins_block`, spanning lines 2-6 and 7 of Figure 3.6(b), directly correspond to the two statements in lines 2 and 3 of Figure 3.6(a) respectively. When executing the instrumented code, arguments of `ins_block` will be evaluated one at a time, evaluating to their models. The models of these statements will be conjoined into the extracted block by the `ins_block` call.

The assignment in line 2 of Figure 3.6(a) is transformed into the sequence of statements in lines 3-5 of Figure 3.6(b), as described by Rule 6 in Table 3.1.

The statement in line 3 of Figure 3.6(b) instruments and evaluates the expression that is being assigned to a variable, storing the resulting model in a temporary variable. In this case, the expression is invoking the `new` method on the `Article` class. This is a core Rails method that creates a new model object, and as such, returns the model of the object creation operation: `create(Article)`.

The statement in line 4 of Figure 3.6(b) assigns a model of a variable read operation to `a`. This value mimics the type that was assigned to the variable in the original program, but otherwise has no state. Whenever any subsequent statement reads variable `a`, the value it reads will be the model of the operation of reading `a`.

The statement in line 5 creates a model representation of an assignment operation - the `Assign` statement node. This node will be returned to the `ins_block` call in lines 1-8 of Figure 3.6(b) as the first argument, ultimately becoming the first child of the `Block` node present in Figure 3.6(c).

The method call in line 3 of Figure 3.6(a) is transformed into the method call in line 7

of Figure 3.6(b). This is in concordance with Rule 2 presented in Table 3.1. The `ins_call` statement will execute a method in an instrumented environment in two steps:

1. Find the method that will be invoked on the provided object with any provided arguments, and instrument it.

2. Invoke the method with any provided arguments.

By default in Rails, the `destroy!` method deletes an object from the database. During instrumented execution, calling `destroy!` on an Article object returns the model of a `delete` operation invoked on the called object. In this case, given that 'a' contains the model of the variable read operation, invoking the `destroy!` method in line 7 results in the `Delete` node present in Figure 3.6(c).

Finally, the models of these statements (`Assign` and `Delete` nodes) are passed to `ins_block`. This method will merge the arguments into a `Block` node that represents the extracted action.

3.2.3 Limitations

Symbolic model extraction has limitations. For one, the approach assumes that the application does not utilize input dependent dynamic features. If it does, for example if a user is given the ability to enter code that will be executed by the application, symbolic model extraction will not extract a sound abstraction of the original source code. In practice, generating code from user input is avoided for performance and security reasons. We encountered this issue only once: in FatFreeCRM, one set of actions determines the class that they are about to handle from a user-supplied string parameter. Since we do not provide this string during symbolic extraction, extraction halts upon encountering this issue.

Our treatment of branches is designed under the assumption that different paths in the program will not use mutually conflicting code generation. Consider a branch statement that executes statement A if the condition holds true and statement B if the condition holds false. Let A and B generate a method under the same name with different source codes. Inside A you would see A's method during both concrete and symbolic execution, inside B you would see B's method during both executions. However, after the branch, symbolic model extraction would only consider B's implementation. Although this is a problem in theory, in practice, we did not encounter such programs. We believe that this problem can be avoided by keeping track of every generated method and using aliasing to access different versions of the same method.

3.3 Symbolic Extraction for Data Model Verification

The first phase of running instrumented execution on a Rails application is to install and configure the analyzed application. To a properly setup application we add our own symbolic model extraction library that overrides core Rails methods with their symbolic versions. Finally, we start model extraction of each action by generating http requests that will invoke them one at the time.

Because ActiveRecord is used in Rails applications to manage data from the database, the usage of ActiveRecord methods and classes is key to extract a model of a Rails application. As such, we override ActiveRecord methods with their symbolic counterparts.

Table 3.2 shows parts of the target modeling languages that are common to the models we used to verify data integrity or access control. Table 3.2 a) contains class (static) methods, and Table 3.2 b) contains instance (object) methods. The first column represents various Ruby on Rails methods. The second column explains the semantics of the corresponding method. The third column defines the statement nodes that are

ActiveRecord method	Semantics	ADSL Statement Node
Class.new(attrs)	Creates an object with provided attributes (basic type values)	create(*Class*)
Class.all	Load all model objects of this type from the database	allof(*Class*)
Class.where(...)	Load all model objects in the database that satisfy some criteria	subset(*Class*)
Class.find(id)	Finds an object using the provided unique identifier	oneof(*Class*)

a) Class methods

ActiveRecord method	Semantics	ADSL Statement Node
expr.select(...)	Returns all objects in *expr* that meet some criterion	subset(*expr*)
expr.association	Returns object(s) related to *expr* via the association	*expr*.association
expr.association << expr2	Associates object *expr* with *expr2* via association *association*	createTuple(*expr*, association, *expr2*)
expr.association = expr2	Mutates an association	*expr*.association = *expr2*
expr.delete!	Deletes the object	delete(*expr*)
expr.destroy!	Deletes the object, propagating deletion to associated objects	delete(*expr*.assoc); delete(*expr*)
expr.destroy_all!	Deletes all objects in a collection expression	delete(*expr*)
expr.each(block)	Executes *block* once for each element in *expr*	foreach v in *expr*: *block*
expr.nil?	Checks whether *expr* is null or not	isempty(*expr*)
expr.any?	Checks whether *expr* has at least one object	not(isempty(*expr*))

b) Instance methods

Table 3.2: ActiveRecord methods and corresponding ADS statement nodes

extracted from the method. This list is not exhaustive because many methods in Ruby on Rails have multiple aliases (different names that achieve the same functionality) for developer convenience.

For example, `Model.new(attrs)` is a constructor. Developers can use this method to create a new object of type *Model*, setting the newly created object's fields corresponding to the *attrs* argument. Similarly, `Model.all` will return a collection of all objects of type *Model* that exist in the database.

Our library will, when the Rails application is booting up, replace core ActiveRecord methods with their instrumented versions. However, other libraries that build on top of ActiveRecord do not need to be manually specified and overridden, as when they implement their functionality on top of core ActiveRecord, they become implicitly prepared for model extraction.

After overriding the core ActiveRecord methods, we identify the set of actions that the Rails application contains. We instrument them and execute them one at the time. Each action will return the model of itself, and these models make part of the entire model we extract from the applications.

For data integrity verification, we make an additional step to extract invariants. These invariants are written in Ruby using a library we developed for this purpose. Adapting our extraction method to extract invariants was straightforward. For example, we added quantification to ActiveRecord objects that can be used to quantify over sets.

3.3.1 Extraction from CanCan

Extraction of CanCan policies requires an additional extraction step to extract the set of user roles and the set of permits.

User Role Extraction

Neither Rails nor CanCan implement authentication by default. Instead, they rely on third party libraries to define the authenticable class and roles. CanCan is usually paired with Devise [29] for this purpose. Extracting the authenticable class and roles is straightforward from an application that uses Devise. If an application does not use Devise, we rely on the convention that the authenticable class is called `User` and define roles according to branch conditions in the Ability object (see Section 3.3.1).

Access Control Policy Extraction

In CanCan, the access control policy is declared in the Ability class. For ease of reading, we repeat the Ability class from Figure 2.1 in Figure 3.7. Every time an action is executed, an Ability object is implicitly generated with the current user in mind.

```
19  ...
20  class Ability
21    def initialize(user)
22      can [:index, :show], :all
23      if user.admin?
24        can :manage, :all
25      else
26        can :manage, User, id: user.id
27        can :manage, Project, user_id: user.id
28        can :manage, Todo, user_id: user.id
29        can :manage, Note
30      end
31    end
32  end
33  ...
```

Figure 3.7: Ability class from Figure 2.1

A typical Ability class constructor is defined with a sequence of `if/elsif/else` branches where branch conditions query the user role of the current user. `can` statements outside these branches apply to all roles. Each `can` statement permits the current user to execute a set of operations on a class type, with optional qualifiers that restrict the set of objects the `can` statement applies to.

In order to extract the policy, we instrument the Ability constructor as follows. Branches are instrumented to execute both paths as usual, but in addition, if the branch condition tests the role of the current user, this is taken note of. Any `can` statements executed in a particular branch will then refer to the corresponding role or roles. In addition, we override the `can` method to generate permits ADS nodes based on its arguments. Finally, we initiate instrumented execution by creating an Ability object, using a `CurrentUser` statement node as the argument.

In order to extract the set of roles a `can` statement applies to, during instrumented execution of the Ability constructor, each branch is associated with a set of roles. Any `can` statement executed under a branch is assigned roles that correspond to the branch condition (or branch conditions, in case branches are nested). The root block of the constructor applies to the entire set of roles.

For example, the `can` statement in line 22 of Figure 3.7 is in the root block of the constructor and, as such, applies to both roles. In line 23 we have a branch condition that only accepts `admin`s, meaning that the statement in line 24 only applies to `admin`s. Statements in lines 26-29 are part of the else branch, which means they refer to all roles associated with the root block but that were not expected by the branch condition: $\{\texttt{admin}, \texttt{nonadmin}\} \setminus \{\texttt{admin}\} = \{\texttt{nonadmin}\}$.

The second piece of information that needs to be extracted is the set of operations. The operation symbols used in `can` statements, by convention, correspond to action names. Moreover, Rails has a strong convention on action names that correspond to create/read/update/delete (CRUD) operations [88]: each model class typically has a separate action for each CRUD operation. Therefore, if we recognize that a user is permitted to execute an action that by convention corresponds to a CRUD operation on objects of a model class, then we infer that the user is permitted to execute the corresponding CRUD operation on objects of the corresponding model class.

Table 3.3 presents our mapping of action names to operations. For example, `new` and `create` actions by convention serve to create a new object of a given model class. If a user is permitted to execute these actions, we infer that the user is permitted to create objects of corresponding type in general.

Operation symbols	Implied CRUD operations
`:manage`	create, delete, read
`:create`, `:new`	create
`:destroy`	delete
`:index`, `:show`	read
`:update`	create, delete

Table 3.3: Mapping actions to CRUD operations

Finally, extracting the expression of a permit is straightforward. If a `can` statement refers to a class without any additional constraints, it applies to all objects of said class in a given data store state. For example, line 10 of Figure 3.7 refers to all objects of the `Note` class, corresponding to an `AllOf('Note')` ADS node.

In case there are additional constraints (such as in lines 26-28 of Figure 3.7), we can use already instrumented methods of ActiveRecord classes to extract the symbolic representation of the set of objects the `can` statement refers to.

To summarize the entire extraction of an ADS permit from the `can` statement in line 22 of Figure 3.7. This `can` statement is executed in the root block and as such applies to all user roles (`admin` and `nonadmin`). It lets users of these roles to execute `:index` and `:show` operations, implying the *read* operation in the resulting permit. It refers to objects of all classes without any additional limitations. Since there are four classes in our running example, Therefore, this `can` statement translates to the following permits:

$$p_1 = \langle \{\texttt{admin}, \texttt{nonadmin}\}, \{read\}, \texttt{AllOf}(\texttt{User}) \rangle$$

$$p_2 = \langle \{\texttt{admin}, \texttt{nonadmin}\}, \{read\}, \texttt{AllOf}(\texttt{Project}) \rangle$$

$$p_3 = \langle \{\texttt{admin}, \texttt{nonadmin}\}, \{read\}, \texttt{AllOf}(\texttt{Todo}) \rangle$$

$$p_4 = \langle \{\texttt{admin}, \texttt{nonadmin}\}, \{read\}, \texttt{AllOf}(\texttt{Note}) \rangle$$

This remaining permits are extracted from the remainder of the Ability object. Permits p_5, p_6, p_7 and p_8 are extracted from the `can` statement in line 24 of Figure 3.7.

Permits p_9, p_{10}, p_{11} and p_{12} are extracted from lines 26, 27, 28 and 29 respectfully.

$$p_5 = \langle \{\texttt{admin}\}, \{read, create, delete\}, \texttt{AllOf}(\texttt{User}) \rangle$$

$$p_6 = \langle \{\texttt{admin}\}, \{read, create, delete\}, \texttt{AllOf}(\texttt{Project}) \rangle$$

$$p_7 = \langle \{\texttt{admin}\}, \{read, create, delete\}, \texttt{AllOf}(\texttt{Todo}) \rangle$$

$$p_8 = \langle \{\texttt{admin}\}, \{read, create, delete\}, \texttt{AllOf}(\texttt{Note}) \rangle$$

$$p_9 = \langle \{\texttt{nonadmin}\}, \{read, create, delete\}, \texttt{CurrentUser} \rangle$$

$$p_{10} = \langle \{\texttt{nonadmin}\}, \{read, create, delete\}, \texttt{Dereference}(\texttt{CurrentUser}, \text{projects}) \rangle$$

$$p_{11} = \langle \{\texttt{nonadmin}\}, \{read, create, delete\}, \texttt{Dereference}(\texttt{CurrentUser}, \text{todos}) \rangle$$

$$p_{12} = \langle \{\texttt{nonadmin}\}, \{read, create, delete\}, \texttt{AllOf}(\texttt{Note}) \rangle$$

Authorization Check Extraction

Extracting the access control policy is not enough for verification of access control in a web application. Runtime enforcement of the policy via authorization checks is an integral part of access control in Rails. Our goal is to ensure that actions, considering implemented authorization checks, correctly enforce the access control policy at all times. There are two ways to check authorization using CanCan: 1) explicitly, using the `can?` method inside action code, and 2) implicitly, using automatically generated authorization checks.

Both explicit and automated authorization checks are extracted during action extraction via instrumented execution, as described below.

Explicit checks. The `can?` method takes two arguments: an *operation* symbol and the *subject* of the authorization check. For example, line 48 of Figure 2.1 (`can? :destroy, @project`) checks if the current user can execute operation `destroy` with the subject being

the object stored in `@project`. The operation is a symbol that matches an operation in the Ability object. The subject of the authorization check might be an object, a set of objects, or a class (denoting all objects of that class). The `can?` method queries the current Ability object to check if at least one `can` statement covers the operation symbol and the subject.

We extract `can?` checks into boolean expressions in our model as follows. First, we extract the subject of the check directly using instrumented execution. Next, we inspect the Ability class to identify all `can` statements that are relevant to the authorization check at hand. Let *ops* be the list of operations checked. For each role r, we identify all `can` statements that apply to role r and whose operations correspond to *ops* and whose expression type corresponds to the type of the subject of the `can?` check. Assuming that $e_1, e_2 \ldots e_k$ are expressions of these `can` statements, we extract `(inusergroup(r) and subject in union(`e_1`, ..., `e_k`))`. We generate this conjunction for each role and disjoin the results to get the complete boolean expression of the `can?` check.

For example, let us extract the `can?` check in line 48 of Figure 2.1. The extracted boolean expression should evaluate to true if and only if the current user is permitted to execute operation `destroy` on the `@project` object.

For each role, we identify `can` statements that are relevant to the check and union their expressions if there are multiple. In this case, we identify `can` statements in lines 24 and 27 of Figure 3.7 The object sets that correspond to these `can` statements and the `Project` class are `allof('Project')` for the `admin` role, and `Dereference(CurrentUser, 'projects')` for the `nonadmin` role.

At this point we have defined the set of objects each role is permitted to operate on. As such, the instrumented `can?` statement returns the boolean ADS statement node in Figure 3.8.

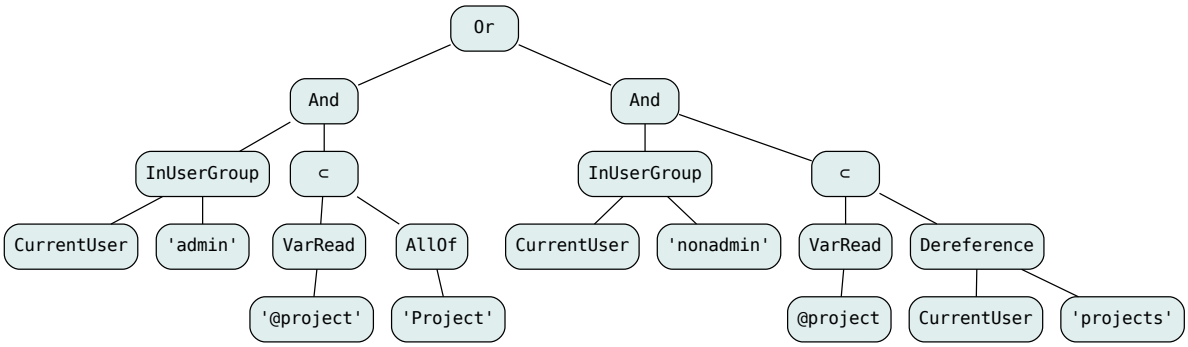


Figure 3.8: Model extracted from the authorization check in line 48 of Figure 3.7.

Automated checks. CanCan can automatically generate access control checks using controller-level declarations that prepend authorization checks to actions. In our experience, most access control checks are created using these declarations. Since these checks are "automagical," we found bugs related to misuse or misunderstanding of these automated checks.

Since we use instrumented execution for model extraction, we do not need to treat automated checks differently from explicit checks. We extract automated checks by allowing CanCan to follow its own logic and heuristics to determine which checks need to be executed, then we extract these generated checks as they are executed.

3.4 Experiments

We used symbolic model extraction to extract models from Rails applications in order to verify their data integrity [12, 13, 15] and access control [16] properties. The result of our experiments are summarized in Table 3.4.

We analyzed a total of 19 open source Rails applications. We found these applications from various sources. We looked at the 25 most starred open-source Rails applications on github according to the OpenSourceRails.com website [83], a compilation of open source

Application	LoC (Ruby)	Classes	Actions	Invariants	Access Control	Permits	Extraction Time (sec)
Avare	1137	6	26	3	✓	34	3.708
Bootstrap	785	2	4	-	✓	2	2.861
Communautaire	753	5	28	6	✓	10	3.236
Copycopter	3201	6	11	6			3.534
CoRM	7745	39	163	32	✓	86	33.868
FatFreeCRM	20178	32	120	8	✓	34	14.383
Fulcrum	3066	5	40	6			4.966
Illyan	1486	3	24	-	✓	4	5.099
Kandan	1535	5	25	6	✓	7	5.395
Lobsters	5501	17	86	9			7.576
Obtvse2	828	2	13	1			6.266
Quant	4124	9	38	4	✓	9	5.688
Redmine	84770	74	264	21			62.295
S2L	1334	9	44	4	✓	15	3.913
Sprintapp	3042	15	120	8	✓	11	14.899
Squash	15801	19	46	18			8.251
Tracks	17562	11	117	9			16.349
Trado	10083	33	66	10	✓	36	12.094
WM-app	2425	18	95	4	✓	36	6.095
Totals	185356	310	1330	155		284	220.476

Table 3.4: Experimental results for symbolic extraction

Rails applications categorized by domain [1], and applications investigated by related work.

Our implementation of symbolic extraction does not support all versions of Rails, as that would require a substantial engineering effort. Our tool supports Rails 3, up to and including Rails 4.2. Furthermore, since we focused on how applications employ ActiveRecord, we did not extract models from applications that bypass ActiveRecord: for example, if they are not backed by a relational database.

We feel that these applications are representative of real world Rails applications for several reasons. They vary in size and complexity, their domain of purpose, the number of developers who developed and maintained them, as well as the technologies they utilize.

For example, FatFreeCRM is an application for customer-relation management. The coding style is characterized by high flexibility in the model, where often the types and names of objects and associations are generated using string manipulations instead of static declarations. That way the same implementation of an action can be used for different controllers and actions. Because the extraction was done in instrumented runtime,

the translator had sufficient information to automatically infer the correct semantics of these actions including the dynamically generated code, which would not be doable by pure static analysis.

Tracks is an application for organizing tasks. It is interesting because of an unorthodox pattern in the data model, where the association among the Todo objects defines a partial order.

Kandan is a chat application that heavily relies on third-party gems, most notably for authentication and authorization (devise[29] and cancan[19]). These libraries offer many features that are manually implemented in Tracks and FatFreeCRM, and therefore, much Kandan's application logic is handled by external gems.

Column *LoC (Ruby)* shows the number of Ruby lines of code in these applications. This number does not include JavaScript, html, dynamic html generation through irb files, or configuration files. Columns *Classes*, *Actions* and *Invariants* show the number of model classes, actions, and invariants respectfully. As invariants are not part of the core Rails framework, we wrote them manually for each application after investigating their source code. We did not write any invariants for Bootstrap and Illyan because their models were too simple to warrant any non-trivial invariants.

Column *Access Control* shows whether the application employs access control through CanCan [20]. For applications that do, we extracted access control information in addition to data integrity. Column *Permits* shows the number of permits extracted from these applications.

Finally, column *Extraction Time* shows the total amount of time it took to extract models from these applications. This includes booting up the Rails application, instrumenting it, and executing each action as described in Section 3.3. We obtained these results on a computer with an Intel Core i5-2400S processor, running 64bit Linux. Memory consumption was typical for starting and running a Rails application.

We manually investigated the extracted models. Based on our manual inspection of the models and the automated verification results we report in Chapter 4, we find the performance and the quality of our model extraction to be acceptable for real world use, potentially both as part of verification of real world applications during the quality assurance process, and for daily developer use.

Chapter 4

Verification via First Order Logic

In this chapter we will first present the translation of ADS language specifications to classical first order logic and show how this translation can be used to verify whether invariants hold on a given ADS. In Chapters 5 and 6 we modify the translation presented here in order to make the translation more conducive to verification.

4.1 Classical First Order Logic

A FOL *language* L is a tuple $\langle F, P, V \rangle$ where F is a set of *function* symbols, P is a set of *predicate* symbols, V is a set of *variable* symbols. All function and predicate symbols are associated with their *arities*, which are positive integers denoting the number of arguments they accept[1].

Given a FOL language $L = \langle F, P, V \rangle$, a *term* is a variable $v \in V$ or a function invocation $f(t_1, t_2 \ldots t_k)$ where $f \in F$ and $t_1 \ldots t_k$ are terms and k is the arity of function f.

[1]We may extend this to introduce constants as functions of arity 0 and propositional variables as predicates of arity 0.

A (well formed) FOL *formula* is defined as either:

- $p(t_1, \ldots t_k)$, where $p \in P$ is a predicate of arity k and $t_1 \ldots t_k$ are terms
- $\forall v\colon f$, where $v \in V$ and f is a formula
- $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ where f_1 and f_2 are formulas
- $t_1 = t_2$, where t_1 and t_2 are terms.[2]

Given a FOL language L, a *structure* S is an instance that may or may not satisfy a formula expressed in this language. More formally, it is a tuple $\langle U, F^S, P^S, V^S \rangle$ where U is a non-empty set of elements called the *universe*. F^S is a mapping of F onto a set of functions such that for every $f \in F$ of cardinality k there exists an $f^S \in F^S$ such that f^S is a function that maps $U^k \to U$. Similarly, for every predicate $p \in P$ of arity k, there exists a $p^U \in P^U$ such that $p^U \subset U^k$.

We can test whether a structure S satisfies a formula (whether the formula is *true* within this structure). To do this we assign elements of U to all terms in the formula. Each variable $v \in V$ is assigned an element $v^S \in U$. Term $f(t_1 \ldots t_k)$ is mapped to the return value of f^U when using elements of U assigned to terms $t_1 \ldots t_k$ as arguments. Similarly, $p(t_1, \ldots t_k)$ is considered to be true if and only if elements corresponding to $t_1 \ldots t_k$ form a tuple that is in P^U. Boolean operators and equality are interpreted in a standard way. Universal quantification is a bit more involved: $\forall v\colon f$ is satisfied by S if and only if, for every structure $S_{(v|e)}$ that is identical to S except that v was assigned a (potentially different) element e of U, f is satisfied by $S_{(v|e)}$.

A formula that is satisfied by one structure may not be satisfied by another. For example, $x = y$ is true for all structures that happen to map variables x and y to the

[2]Although classical FOL does not include equality, since the theorem provers we use operate on FOL with equality, we include equality in our definition of FOL.

same element. A formula $\forall x\colon (\forall y\colon x = y)$ is true if and only if U is a singleton set. If a formula is satisfied by all structures, we call this formula *valid*. E.g. $x = x$ is a valid formula.

We take note of *free variables*: variables that are not quantified outside the term in which they appear. For example, $\forall x\colon x = y$ has one free variable y. Since theorem provers we use do not allow free variables, from this point on, we will only evaluate the truth value of formulas without free variables. Such a formula is true if and only if it is valid for all structures.

4.2 ADS Translation

To translate an ADS to first order logic for verification, we create a different set of formulas for each action and for each invariant. We translate the schema (Section 4.2.1), the action (Section 4.2.2), and invariants in the pre-state (Section 4.2.3) into axioms. We also translate the invariant in question into a conjecture. If the resulting set of axioms implies the conjecture, then the action correctly preserves the invariant. If axioms do not imply the conjecture, a bug is reported as there exists a way for the action to invalidate the invariant that is being verified. This recipe is changed depending on the theorem prover (further discussed in Chapter 6), but this explanation suffices for the purpose of this chapter.

A single translation that models all actions and all verifies all invariants at once is feasible, but we decided to not take this approach for two reasons. First, this would make identifying a detected bug difficult, as the theorem prover would show that an action could break an invariant without specifying which invariant and which action are the violators. Second, the resulting set of formulas would be rather large and if a theorem prover were not able to terminate for any isolated action/invariant pair, it

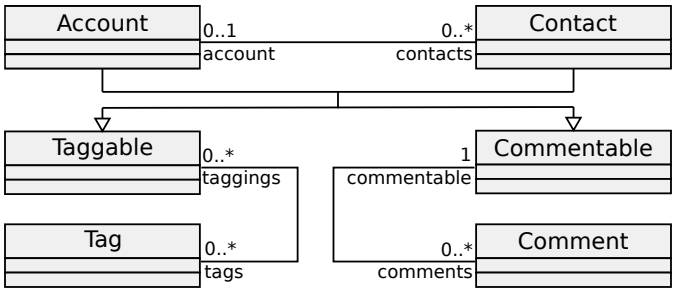


Figure 4.1: A data model schema example based on FatFreeCRM [36]

would probably not terminate if given all actions and invariants (depending on theorem prover heuristics, it may be possible that this would instead terminate, though extremely unlikely). Such a failure would provide no partial result to the developer. By partitioning the problem and verifying each action/invariant or authorization property in isolation, the developer can get results for everything successfully proven or falsified even if there exist action/invariant pairs or authorization properties for which the theorem prover produced no conclusive result.

In this section we frequently conjoin or disjoin a set of formulas. When a set of conjoined or disjointed formulas is empty, we substitute the conjunction or disjunction with their neutral elements (*true* and *false* respectively).

4.2.1 ADS Schema Translation

We will use the class diagram in Figure 4.1 as the running example in this subsection. We assume to be given a data store $DS = \langle C, L, A, I, R, P \rangle$.

Class translation. First, for each class $c \in C$, we define a unary predicate $\overline{c}$ that semantically denotes whether its argument represents either an instance of that particular class or an instance of any superclass.

Then we define axioms that enforce our type system. We define three groups of

Predicates: Account, Contact, Taggable, Commentable, Tag, Comment, XTaggable, XCommentable.

$\forall o\colon \texttt{Account}(o) \rightarrow \texttt{Taggable}(o) \wedge \texttt{Commentable}(o)$ (1)

$\forall o\colon \texttt{Contact}(o) \rightarrow \texttt{Taggable}(o) \wedge \texttt{Commentable}(o)$ (2)

$\forall o\colon \texttt{XTaggable}(o) \leftrightarrow \texttt{Taggable}(o) \wedge \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o)$ (3)

$\forall o\colon \texttt{XCommentable}(o) \leftrightarrow \texttt{Commentable}(o) \wedge \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o)$ (4)

$\forall o\colon \texttt{Account}(o) \rightarrow \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o)$ (5)

$\forall o\colon \texttt{Contact}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o)$ (6)

$\forall o\colon \texttt{XTaggable}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o)$ (7)

$\forall o\colon \texttt{XCommentable}(o) \rightarrow \neg\texttt{Account}(o) \vee \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o)$ (8)

$\forall o\colon \texttt{Tag}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Comment}(o)$ (9)

$\forall o\colon \texttt{Comment}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o)$ (10)

Figure 4.2: Axioms defining the class diagram in Figure 4.1 in classical FOL

axioms: *inheritance axioms* that define superclass relationships, *instance axioms* that define predicates that we can use to denote that an object is an instance of a given class (specifically not of a subclass), and *membership axioms* that define that every object is an instance of at most one class.

Inheritance axioms define that objects of subclass types are also of superclass types. For each class $c \in C$ that has a non-empty superclass set $\text{superclass}(c) = \{p_1, p_2 \dots p_k\}$ we generate an axiom:

$$\forall o\colon \overline{c}(o) \rightarrow \overline{p_1}(o) \wedge \overline{p_2}(o) \wedge \cdots \wedge \overline{p_k}(o)$$

For example, given the model in Figure 4.1 this method produces Formulas (1) and (2) in Figure 4.2.

Instance axioms constitute one axiom per class $c \in C$ and serve to define *instance predicates* $\overline{c}_x$. These predicates are used to express that an object is an instance of class c, or more explicitly, of class c but not of any of c's subclasses. Given $\{s_1 \dots s_k\}$, the set of all direct subclasses of c (all classes s for which $c \in \text{superclass}(s)$), we generate an

axiom:

$$\forall o\colon \overline{c}_x(o) \leftrightarrow \overline{c}(o) \wedge \neg\overline{s_1}(o) \wedge \cdots \wedge \neg\overline{s_k}(o)$$

Note that, if c has no subclasses, this axiom defines equivalence between $\overline{c}$ and $\overline{c}_x$. If this is the case, as an optimization, we omit defining $\overline{c}_x$ and use $\overline{c}$ instead. Given the model in Figure 4.1 this creates Formulas (3) and (4) in Figure 4.2.

Membership axioms define that each object represents an instance of exactly one class. Assuming that $C = \{c_1 \dots c_k\}$, for every class $c_i \in C$, we create an axiom in order to constrain that, if an object is an instance of class c_i, it cannot be an instance of any other class:

$$\forall o\colon \overline{c_i}_x(o) \rightarrow \neg\overline{c_1}_x(o) \wedge \cdots \wedge \neg\overline{c_{i-1}}_x(o) \wedge \neg\overline{c_{i+1}}_x(o) \wedge \cdots \wedge \neg\overline{c_k}_x(o)$$

These formulas correspond to Formulas (5)-(10) in Figure 4.2.

The resulting number of generated formulas is linear in the number of classes, and so is the size of these formulas.

Association translation. Similarly to objects, we use FOL universe elements to represent tuples. A convenient consequence of this approach is that it allows us to define the creation and deletion of objects and tuples uniformly. We introduce unary predicates *is_object* and *is_tuple* to distinguish whether a universe element represents an object or a tuple. We define that no domain element can be both an object and a tuple.

For each association $l \in L$ we introduce a unary predicate $l(t)$ that returns true if and only if t is representing a tuple that belongs to l.

In order to associate tuples with objects, for each association $l \in L$ we define two unary functions: $origin_l(t)$ and $target_l(t)$ such that $t = \langle l, origin_l(t), target_l(t)\rangle$.

We enforce association cardinality constraints using formulas to limit the number of tuples per origin/target object in a data store state. Note that we do not enforce cardinality globally, but only in the action's pre and post state. We do this because real world applications often invalidate cardinality temporarily while an action is executing.

4.2.2 Action translation

Actions are the most complex part of ADS translation to FOL. We will first define how states are represented in FOL, then define how object sets and boolean nodes are translated to FOL. Only then do we have all the definitions necessary to define statements.

For brevity, we will not present the translation of all ADS nodes in Tables 2.1 and 2.2. Here we only include the definitions of the most representative or interesting nodes.

States are translated to unary predicates that define which objects and tuples exist in the state. For example, given a state predicate s, if $s(x)$ then x is a domain element representing either an object or a tuple that exists in state s.

Object set translation

Most ADS statement nodes manipulate or return object sets. An object set α represents a set of objects that share a common set of classes or superclasses.

Every object set α is translated into a formula F_α that has one free variable o. Object sets inside loops are translated to formulas with more than one free variable, but for simplicity, we will focus on object sets outside loops in the following discussion.

F_α evaluates to true if and only if the free variable o is assigned a universe element that represents an object that belongs to the object set α. Object set formulas are meant to be directly injected into formulas that quantify over these free variables, and use the object set according to the statement's semantics.

For example, let us translate an object set node `AllOf(c)` to FOL. Let c be the type predicate corresponding to the argument class, and let s be a predicate denoting the state in which the object set is being evaluated. Then, the object set is defined simply using the formula:

$$s(o) \wedge c(o)$$

meaning that domain element o represents a member of this object set if and only if it exists in the current state s and is of class c. Note that o is the one free variable in this formula.

As an another example, a `Subset` node semantically evaluates to an object set that is a subset of its argument object set. Let β be the argument object set of a subset node α. To translate a `Subset` node in state s, we introduce a new predicate $subset_\alpha(x)$ and define an axiom:

$$\forall x \colon subset_\alpha(x) \Rightarrow F_\beta(x)$$

With this axiom, $F_\alpha(o)$ is translated as:

$$subset_\alpha(o)$$

The resulting formula still has o as a free variable, meaning that it is a valid object set formula. Notice that we enforced this subset function to be non-deterministic by not having any rules on which o is included in the subset using $subset_\alpha(o)$.

More interesting are `Assign` and `VarRead` nodes. Note that this translation is done after transforming the program to static single assignment that ensures that all variables are defined once.

The `Assign` node takes two arguments: a variable identifier v and object set node α. It defines a predicate v that corresponds to the object set set α. It accomplishes this by defining an axiom:

$$\forall x\colon v(x) \Leftrightarrow alpha(x)$$

After defining this predicate, the `Assign` node returns $v(o)$ as its object set formula.

Variable read nodes `VarRead` take a variable v as an argument. When evaluated in state s, they translate to the object set formula denoting all object that were assigned to v and still exist in the current state:

$$v(x) \wedge s(x)$$

Boolean expression translation

Boolean expressions are translated to formulas that are embedded into statements, similarly to object sets. Unlike object set formulas, these formulas do not have free variables, unless inside a loop. Free variables are only introduced to facilitate loops, as will be explained further down in this section.

For example, the `IsEmpty` statement node takes an object set as an argument and evaluates to *true* if and only if the argument object set is empty. If object set α is this argument, the node translates to formula:

$$\forall x\colon \neg\alpha(x)$$

As one more example, the $\subset$ node accepts two object sets and is expected to evaluate to *true* if and only if the first object set is a superset of the other. Let $F_1(o)$ and $F_2(o)$ be

the translations of the two object sets. The $\subset$ node translates to the formula:

$$\forall o\colon F_1(o) \rightarrow F_2(o)$$

Translation of state migrations

Many ADS statement nodes migrate the data store state. Each state migration can be represented as a set of pairs of states $\langle s, s' \rangle \subseteq \overline{DS} \times \overline{DS}$ which semantically represent possible state transitions by means of that statement node. For a statement node S and two states s and s', we will use $[s, s']_S$ to denote that $\langle s, s' \rangle$ is a possible execution (state transition) of S.

For example, let us translate a `Delete` statement node D that deletes objects from an object set α, as well as all tuples associated with deleted objects. Assuming there exists only one association between class t and other types, this statement node can be translated as follows:

$$\begin{aligned}
&\forall s, s' \in \overline{DS} \times \overline{DS}\colon \\
&\quad [s, s']_D \Leftrightarrow (\forall o\colon \text{is_object}(o) \Rightarrow (o \in s' \Leftrightarrow o \in s \wedge \neg F_\alpha(o))) \\
&\quad \wedge (\forall t\colon \text{is_tuple}(t) \Rightarrow t \in s' \Leftrightarrow (t \in s \wedge \neg(F_\alpha(\text{origin}_r(t)) \vee F_\alpha(\text{target}_r(t)))))
\end{aligned}$$

As an other example, consider a `Block` statement node B, which contains a sequence of other statement nodes A_i for $1 \leq i \leq n$ for some n. Let statement node A_1 transition between states s and s_1 if and only if $[s, s_1]_{A_1}$. The set of states that the sequence $A_1; A_2$ can transition to from s is equal to the union of all states that A_2 can transition to from any state s_1 such that $[s, s_1]_{A_1}$. Therefore, $\forall s, s' \in \overline{DS} \times \overline{DS} : [s, s']_{A_1;A_2} \Leftrightarrow (\exists s_1 \in \overline{DS} :$

$[s, s_1]_{A_1} \wedge [s_1, s']_{A_2}$). If we extrapolate this reasoning to the whole block B:

$$\forall s, s' \in \overline{DS} \times \overline{DS} : [s, s']_B \Leftrightarrow (\exists s_1, s_2 \dots s_{n-1} \in \overline{DS} \times \dots \times \overline{DS} : \\ [s, s_1]_{A_1} \wedge [s_1, s_2]_{A_2} \wedge \dots \wedge [s_{n-1}, s']_{A_n})$$

Loop translation. A `ForEach` loop statement (FE) is defined with three parameters: the set of objects being iterated over, the variable containing the iterated value, and the block of code that will be executed for each object in the object set. Let α be the object set, v the variable, and B the block of code. Let $|\alpha| = n$. By definition, the order of iteration is non-deterministic.

Since B has access to the iterated object that is different for each iteration, executions of B are affected by the iterated variable. Effectively, each iteration is a different state transition: we use notation $[s, s']_{B_o}$ to refer to a possible execution of an iteration executed for object o. In this case, we refer to o as the *trigger object*. The formula defining the FE loop is:

$$\forall s, s' \in \overline{DS} \times \overline{DS} : [s, s']_{FE} \Leftrightarrow \exists o_1 \dots o_n \in \alpha, \exists s_1 \dots s_n \in \overline{DS} : \\ \forall i, j \in [1 \dots n] : i \neq j \Leftrightarrow o_i \neq o_j \wedge \quad (1) \\ [s, s_1]_{B_{o1}} \wedge [s_1, s_2]_{B_{o2}} \wedge \dots \wedge [s_{n-1}, s_n]_{B_{on}} \wedge s_n = s' \quad (2)$$

In words, a pair of states is an execution of a given loop FE if and only if there exists an enumeration of objects from α and a sequence of states such that (1) the said enumeration of objects is a permutation of α, and (2) the said sequence of states is achievable by triggering iterations in the order of the object permutation.

There exists a corner case where an object that is about to trigger an iteration gets deleted by a prior iteration. We did not include this corner case as part of the definition as

it introduces considerable complexity, but the semantic is as follows: such an iteration will still execute with an empty set iterator variable value. This behavior is in concordance with our abstraction and the behavior of ORM tools when objects are deleted before triggering iterations.

We take a closer look at the translation of loops in Chapter 5, and improve on it in order to better facilitate verification.

4.2.3 Invariant Translation

Invariants are translated as boolean expressions. Unlike boolean expressions, invariants can be translated twice: once in the pre-state of the action and, for the purpose of data integrity verification, once in the post-state of the action. In the pre-state of an action, we state that the conjunction of all invariants holds, defining that the pre-state is consistent. We translate the invariant once again in the post-state if we are verifying data integrity.

4.2.4 Access Control Verification

We want to use FOL theorem provers to verify that, regardless of the role of the current user, all objects created, deleted or read by any possible execution of an action are permitted for creation, deletion and reading, respectively. We generate a set of *authorization properties* to express this expectation.

An authorization property is a tuple $\langle a, op, c \rangle$ that defines the expectation that action $a \in A$ correctly enforces the access control policy with respect to the operation $op \in \{create, read, delete\}$ for all objects of type $c \in C$. We generate an authorization property for every action, every operation, and every class where the action might possibly execute that operation on objects of that class. For example, we will not generate an

authorization property for creating Articles in an action that never creates an Article, as this authorization property is vacuously valid.

Our goal is to, given an authorization property $\langle a, create, c\rangle$, generate a formula that verifies whether the access control policy is correctly enforced with regards to this property. To do this we need to first express the set of objects created, deleted or read by an action in FOL. We also need to translate permits to FOL. With these two, we can express the expectation that all objects created, deleted or read by an action are accepted by at least one permit in FOL.

Create, delete and read sets of an action. As a consequence of translating an action to FOL using existing techniques [12], we have access to a predicate $s_c(o)$ that determines whether object o of class c exists in the action's pre-state s. Similarly, we have access to a predicate $s'_c(o)$ that determines whether object o of class c exists in the action's post state s'. We can use these predicates to define new predicates $created_c$ and $deleted_c$ that identify objects of class c that have been created or deleted by an action:

$$\forall o \in O_c \colon created_c(o) \Leftrightarrow \neg s_c(o) \wedge s'_c(o)$$

$$\forall o \in O_c \colon deleted_c(o) \Leftrightarrow s_c(o) \wedge \neg s'_c(o)$$

In order to identify all objects read by an action, we look for objects stored in any variable whose name begins with `@`. In Rails, these variables are referred to by the view in order to synthesize the response. When translating an action to FOL, after static single assignment, every variable v corresponds to a predicate $v(o)$ that determines whether an object is stored in the variable. Formally, given an action a, class c, and a set of variables

$v_1 \dots v_i \in V$ that are of type c and whose name starts with `c`:

$$\forall o \in O_c \colon read_c(o) \Leftrightarrow s'(o) \wedge (v_1(o) \vee \dots \vee v_i(o))$$

Translation of permits and permit acceptance. Given a permit $p = \langle g, ops, e \rangle$, a specific operation op and state predicate $s(o)$, we can generate a predicate $p_{op,s}(o)$ that accepts objects o if and only if $p[op, s, o]$. This predicate should be satisfied by o if and only if three conditions hold: $op \in ops$, the current user has a role in g, and o belongs to the result of expression e in state s.

The first condition is the easiest to translate. If $op \notin ops$, this permit does not accept o. We can at this point stop translating $p_{op,s}(o)$ to first order logic and declare that it will accept no object o.

The second condition checks the role of the current user. In our FOL translation, each role $r \in \overline{R}$ corresponds to a predicate $r(o)$ where $o \in O_{c_U}$. Therefore, if $g = \{r_1 \dots r_k\}$, the second condition is $r_1(c_U) \vee \dots \vee r_k(c_U)$.

Finally, we need to translate the expression e to FOL. All expressions e, in actions and permits alike, translate to formulas with a single free variable x (denoted as $f(x)$) such that $f(x)$ is satisfied by all x that correspond to objects that the expression evaluates to [12]. Therefore the third condition is directly defined as $e(o)$.

With that in mind, we can define the permit predicate $p_{op,s}$. If $op \notin ops$:

$$\forall o \in O_c \colon \neg p_{op,s}(o)$$

Otherwise:

$$\forall o \in O_c \colon p_{op,s}(o) \Leftrightarrow (r_1(o_U) \vee \dots \vee r_k(o_U)) \wedge e(o)$$

Translation of authorization properties. After defining the create, delete and read sets of actions, as well as translating permits, we have all the tools we need to generate FOL formulas that correspond to authorization properties. In order to check whether an action is valid with regards to authorization property $\langle a, create, c\rangle$, we use the theorem prover to check whether the following formula is implied from the definition of the action. Let the set of permits P be $\{p_1 \dots p_k\}$:

$$\forall o \in created_c(o) \colon p_{1\ create,s'}(o) \vee \dots \vee p_{k\ create,s'}(o)$$

In other words, an action is valid with regards to authorization property $\langle a, create, c\rangle$ if and only if every object of class c created by a is accepted by at least one permit p: $p[create, s', o]$.

We similarly define the condition for whether an action is valid with regards to authorization properties $\langle a, read, c\rangle$ and $\langle a, delete, c\rangle$ as:

$$\forall o \in read_c(o) \colon p_{1\ read,s'}(o) \vee \dots \vee p_{k\ read,s'}(o)$$
$$\forall o \in deleted_c(o) \colon p_{1\ delete,s}(o) \vee \dots \vee p_{k\ delete,s}(o)$$

Note that, in accordance with our discussion in Section 2.3, the condition for $deleted_c(o)$ refers to the pre-state, while the conditions for $created_c(o)$ and $read_c(o)$ both refer to the post-state.

4.3 Experimental Evaluation

We evaluated the tool on 19 open source Rails applications, presented in Table 3.4. Chapter 3 discusses the approach and performance of model extraction, here we focus

Application	Verified Properties	Average Predicates	Max Memory (Mb)	Average Time (sec)	Verified	Falsified	Timeouts	False Positives	Bugs
Avare	111	59	50	0.02	75	36	0	0	9
Bootstrap	4	15	2	0.00	3	1	0	0	1
Communautaire	216	35	3	0.00	208	2	0	6	1
Copycopter	66	47	46	0.01	66	0	0	0	0
Corm	5555	436	93	0.08	5432	88	9	26	35
FatFreeCRM	1134	192	7	0.02	1087	44	0	3	15
Fulcrum	240	40	4	0.01	233	7	0	0	4
Illyan	46	44	2	0.00	36	10	0	0	1
Kandan	173	34	4	0.01	160	13	0	0	4
Lobsters	774	201	69	0.77	753	13	4	4	10
Obtvse2	13	10	1	0.00	13	0	0	0	0
Quant	203	45	2	0.01	203	0	0	0	0
Redmine	5565	432	20	0.05	5542	12	5	6	7
S2L	310	124	6	0.02	230	70	0	10	12
Sprintapp	1079	96	29	0.03	1033	46	0	0	32
SquareSquash	828	175	9	0.03	824	4	0	0	4
Tracks	1053	88	286	0.23	1036	13	3	1	6
Trado	724	278	50	0.43	638	78	2	6	20
WM-app	418	117	5	0.01	418	0	0	0	0
Total	18512	317.38	286	0.11	17990	437	23	62	161

Table 4.1: Verification experiments summary

only on verification of extracted models.

4.3.1 Verification details and identified bugs

Our tool verifies each extracted ADS specification using Spass [107] and Z3 [25] for theorem proving. First we translate action/invariant pair (or authorization property) to FOL. These FOL formulas are sent both to Z3, an SMT solver, and Spass, a FOL theorem prover. We express our FOL theorems in SMT using problem group UF, which includes free quantification, free sorts and uninterpreted functions. Z3 checks satisfiability, so when we construct a formula to be sent to Z3, if a satisfying model exists for the formula, then there exists an execution of the action violates data integrity (or the access control policy). If, on the other hand, Z3 reports that the formula is unsatisfiable, then we can conclude that the action correctly enforces data integrity (or access control). Spass, on the other hand, checks whether a conjecture implies from a set of axioms. This conjecture is a formula that defines that data integrity is preserved (or authorization property correct). If Spass reports that the conjecture implies from the axioms, then

we can conclude that no execution that violates data integrity (or the access control policy) exists. However, if Spass reports that the conjecture does not always imply from the axioms, then we can conclude that an execution that violates data integrity (or the access control policy) exists.

Note that FOL is undecidable in general. We generate formulas without restrictions on quantification nesting, without a bound on the number of arguments for predicates, and without a bound on the domains. The formulas we generate are not in a decidable fragment of FOL that we could find. This implies that Z3 and Spass may not be able to produce a conclusive result for some of the formulas we generate.

We use Z3 and Spass concurrently, waiting for either theorem prover to produce a result, after which the other prover is terminated. In our experiments we observed that Z3 is faster and is more likely to report conclusive results for the formulas we generate. We will compare these theorem provers directly in Section 6.4. In this chapter, we focus on the overall results of our approach.

Table 4.1 shows the verification results we produced. These experiments were run on a computer with an Intel Core i7-6850K processor, with 128GB RAM, running 64bit Linux. We run a total of 12 processes concurrently to cover the experimental set.

Column *Verified Properties* shows the number of action/invariant pairs and authorization properties generated from the application. Each of these properties is translated to FOL and verified independently. Column *Average Predicates* shows the average number of predicates in FOL formulas we generated. Corm and Redmine have the highest number because their schemas are the most complicated, increasing the number of predicates and axioms needed to specify it.

Column *Max Memory (Mb)* shows the maximum memory a theorem used to produce a conclusive result. Column *Average Time (sec)* shows the average time it took before a theorem prover (Z3 or Spass) took to deduce a conclusive result. Columns *Verified*, *Fal-*

sified, *Timeouts* and *False Positives* refer to the number of properties that were verified, falsified, timed out, or that reported a bug that we found not to be an actual bug.

From a total of 18512 properties, we verified 17990 to be correct. 437 of properties failed verification, which we manually traced to actual bugs in the application. 23 properties timed out for both theorem provers, and there were 62 false positives which were manually confirmed not to be actual bugs.

These false positives are due to the impreciseness of our extraction method. Specifically, we do not differentiate objects that are saved to the database from those that are not. Sometimes actions create objects that are not saved to the database, either to provide to the view or to process data, and sometimes these objects would invalidate the database if saved. These false positives could be avoided if the extraction method were made more precise.

Finally, column *Bugs* lists the number of distinct bugs we identified based on falsified properties. Often, addressing one falsified property will fix other falsified properties too. For example, if the access control policy is deficient, fixing the access control policy might fix more than one falsified property (one for each action).

Our criterion for determining which falsified properties correspond to distinct bugs is based on the fix required to address the falsified property. For example, if multiple falsified properties can be fixed with a single controller-level change, we consider all those falsified properties to correspond to the same bug. As another example, if multiple falsified properties can be fixed on the level of the model class, we consider all those falsified properties to refer to the same bug. We manually analyzed all the falsified properties based on this criterion, and based on our analysis, we identified 161 bugs that correspond to 437 falsified properties.

Data integrity bugs. We discussed some of these bugs in Section 1.1. This list is not exhaustive, but it shows a few bugs that we found.

In Tracks, one bug we found showed that it is possible to orphan an instance of a `Dependent` class. This stems from the way the `ProjectController` deletes a `Project`; it cleans up all `Project`'s `Todo`s, but does not clean up the `Dependencies` of deleted `Todo`s. The other bug is caused when a `User` is deleted using the `UsersController`. All `Project`s of the `User` are deleted, but `Note`s of deleted `Project`s remain orphaned.

In FatFreeCRM, one of the bugs we found relates to `Permission` objects. These objects define permissions for either `User` or `Group` objects. Our tool has shown that it was possible to have a `Permission` without associated `User` or `Group` objects. The other bug relates to `Todo` objects being orphaned when a `User` is deleted.

Access control bugs. There are 11 actions in CoRM that violate the access control policy in a total of 26 different ways. In AccountsController and ContactsController, `destroy` actions can be used to, as a side-effect, delete Tasks and Aliases. The policy explicitly forbids the superuser role from deleting these objects, yet superusers are allowed to execute these actions. Furthermore, none of the actions in the Emails controller ever checks policies, accounting for several access control violations. The same holds for the Imports controller mentioned in Section 1.1.

Our tool found 4 access control violations in SprintApp, all ultimately caused by poor communication between CanCan and ActiveAdmin, a library SprintApp uses for action generation. In SprintApp, ActiveAdmin pre-loads data relevant to the action, and CanCan subsequently authorizes operations on this data. However, in four actions, CanCan fails to capture the pre-loaded objects and proceeds to authorize the action ignoring pre-loaded objects. ActiveAdmin then proceeds with the pre-loaded objects that were not correctly authorized. Using this exploit, any user can access the private

information of any user and change their information including their password. What is specifically interesting about this bug is that it is not caused by a developer error, but by a configuration error. The action and the controller source code both look like they authorize data correctly.

There is a total of 10 policy violations in Kandan. The policy does not refer to Attachment and Activity classes, implicitly disallowing operations on them. However, the policy is never checked with regards to these classes. Semantically all operations should be allowed on them, and as such, this represents a deficient policy.

Quant uses a simple access control policy. There is only one user group, and every user can access only their own objects. This policy is simple enough to correctly be enforced using automated checks that are present in every controller. As such, this application has no access control violations.

Trado is an e-commerce platform, letting users browse items, add them to their carts and place orders. However, the policy only mentions a few of these classes, most notably not mentioning Orders and order-related classes at all. In addition, it attempts to enforce this policy in only a few controllers, and as such is enforced inconsistently. For the sake of consistency we categorized these errors as policy errors as the policy is too sparse to be usable, however, even a more complete policy wouldn't be correctly enforced on the application as-is. The policy and the enforcement of the policy are both amateurish, but show that our tool is useful for detecting basic errors in addition to esoteric ones.

Other applications that utilized access control have numerous access control bugs in them, but these applications are typically developed by beginners and abandoned mid-development. Wm-app has no bugs or policy errors, and is a nice example of how automated access control can be used correctly and uniformly. All applications other than Wm-app and Illyan that utilized access control had errors either in the access control policy or access control enforcement.

Chapter 5

Coexecutability

In Chapter 4 we demonstrated that one can check invariants about the data store by translating verification queries about actions to satisfiability queries in First Order Logic (FOL), and then using an automated FOL theorem prover to answer the satisfiability queries. However, due to undecidability of FOL, an automated theorem prover is not guaranteed to come up with a solution every time, and sometimes it may timeout without providing a conclusive result.

In our experience, actions that have loops in them are the most difficult to check automatically. In general, verification of code that contains loops typically requires manual intervention where the developer has to provide a loop invariant in order to help the theorem prover in reasoning about the loop. This reduces the level of automation in the verification process and, hence, its practical applicability. Otherwise, unrolling the loop a finite number of times fully automatic but makes verification unsound.

Automated reasoning about loops is difficult since it is necessary to take into account many possible intermediate states that can appear during the loop execution. In this chapter, we present a fully automated technique that significantly improves the

```
1  class PostsController
2    def destroy_tags
3      ...
4      posts = Post.where(id: params[:post_ids])
5      ...
6      posts.each do |p|
7        p.tags.destroy_all!
8      end
9      ...
10   end
11 end
```

Figure 5.1: An example action

verifiability of actions with loops. Here we define *coexecution*, an alternative definition of iterative loops which, while intuitively similar to parallel or concurrent execution, does not correspond to an execution on actual hardware. It is a concept we introduce specifically to make verification easier. We call a loop *coexecutable* if coexecution of its iterations is equivalent to their sequential execution. We present an automated static analysis technique that determines if a loop is coexecutable. We also developed a customized translation of coexecutable loops to FOL that exploits the coexecution semantics and improves verifiability.

5.1 Coexecution overview

Figure 5.1 presents an example Rails action. This action deletes all `Tag` objects associated with a set of `Post` objects. The set of `Post` objects that are chosen by the user are assigned to a variable called `posts` in line 4. Then, using a loop (lines 6 to 8) that iterates over each `Post` object in `posts` (with the loop variable `p`), all `Tag` objects that are associated with such `p` are deleted.

A data store invariant could be that each `Tag` object in the data store is associated with at least one `Post` object. In order to verify such an invariant, we need to prove that each action that updates the data store (such as the one shown in Figure 5.1) preserves

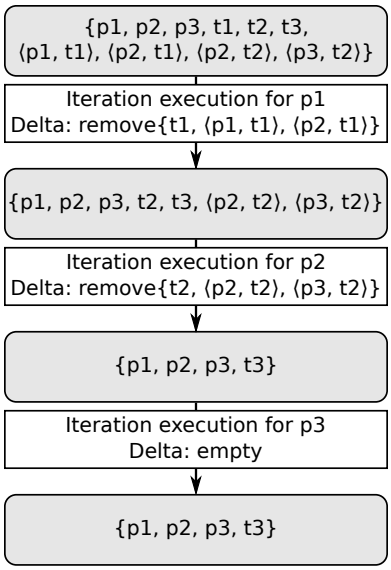


Figure 5.2: Example of sequential execution

the invariant.

Actions with loops are especially hard to verify when translated to FOL. Upon looking at deduction logs of our theorem prover, we noticed that it was often getting stuck while reasoning about states between subsequent iterations. Since the number of iterations is unbounded and theorem provers attempt to enumerate dependencies between iterations, even loops with empty bodies can cause inconclusive results. We observed that most iterative loops seen in web applications can be modeled in a way that does not require iteration interdependency. In such cases, it is possible to translate the loops to FOL in a way that is more amenable to verification. We call this alternate execution model for loops that removes iteration interdependency *coexecution*.

Consider the action shown in Figure 5.1. Assume that the initial data store state contains three `Post` objects $p1$, $p2$ and $p3$ and three `Tag` objects $t1$, $t2$ and $t3$, with the following associations: $\langle p1, t1\rangle, \langle p2, t1\rangle, \langle p2, t2\rangle, \langle p3, t2\rangle$. Also, assume that the loop iterates on all the `Post` objects in the order $p1, p2, p3$ (i.e., the variable `posts` is the ordered collection of $p1, p2, p3$).

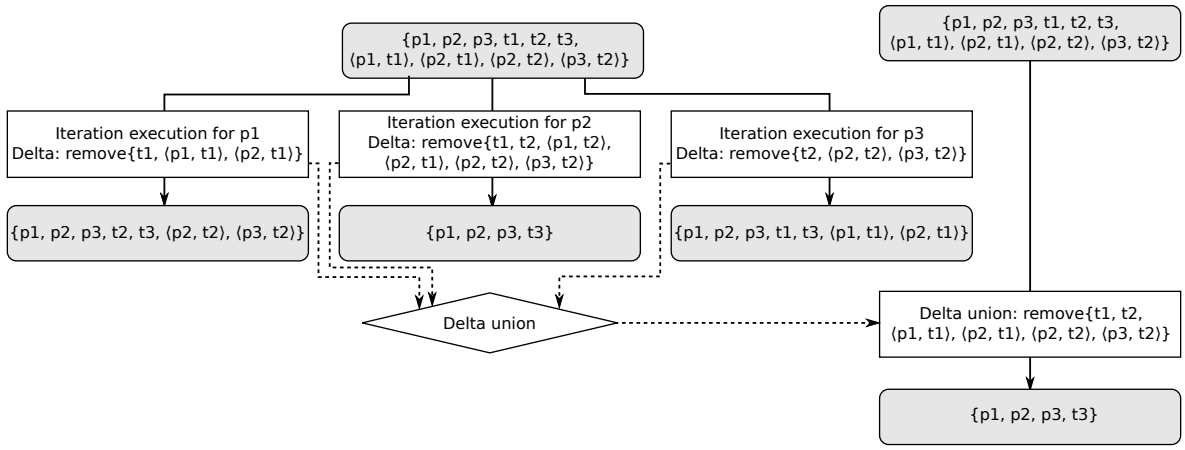


Figure 5.3: Example of coexecution

Given this initial state, the standard sequential execution semantics of the loop in lines 6 to 8 in Figure 5.1 is shown in Figure 5.2 where gray rectangles with rounded corners denote states of the data store, and solid line arrows denote iteration executions.

The first iteration, executed for `Post p1`, deletes all `Tags` of $p1$ and their associations. We identify the operations executed by this iteration and summarize them as a *delta* of this operation (i.e., the changes in the data store state caused by this operation). The first iteration removes $t1$ and all associations of $t1$ ($\langle p1, t1\rangle$ and $\langle p2, t1\rangle$). Similarly, the second iteration deletes all `Tags` of $p2$, resulting in a delta that removes $t2$, $\langle p2, t2\rangle$ and $\langle p3, t2\rangle$. Finally, the third iteration does not alter the data store state as $p3$ is not associated with any `Tags` at that point.

Figure 5.3 demonstrates the alternate *coexecution* semantics for the same loop. Instead of executing iterations sequentially to reach the post-state, we first identify the delta for each iteration directly from the pre-state, in isolation from other iterations. As expected, the first iteration (the one for $p1$) is identical to the one from Figure 5.2. However, the iteration for $p2$ deletes all `Tags` of $p2$, its delta removing $t1$ and $t2$ and all their associations. Note that this delta is different from the delta of the sequential execution iteration for $p2$ shown in Figure 5.2. Similarly, the iteration for $p3$ deletes $t2$ and all

associations whereas sequential execution for $p3$ produced an empty delta. The deltas of these independent executions are combined together using the *delta union* operation, which in this case returns a union of all the delete operations (we formally define the deltas and the delta union operation in Section 5.2). In this example, the unified delta removes $t1$, $t2$ and their associations. Finally, we use the unified delta to migrate from the pre-state to the post-state in one step, reaching the same post-state we acquired using sequential execution as shown in Figure 5.3. We call this one step execution semantics based on the unified delta, coexecution.

For some loops, based on the dependencies among loop iterations, coexecution will yield a different result than sequential execution. However, coexecution is equivalent to sequential execution for some classes of interdependencies. Note that, in our example, iterations *are* interdependent (since the $p1$ iteration prevents the $p2$ iteration from deleting $t1$ and its associations), and yet coexecution and sequential execution produce identical results. In Section 5.2.4, we formally define the *Coexecutability Condition* that, if true for a given loop, guarantees that coexecution of the loop is equivalent to sequential execution of iterations. In Section 5.3 we implement this condition as a static program analysis, and based on this analysis we are able to translate loops to FOL in a manner that is more amenable to verification.

Remember that we assume that actions that update the data store are executed as transactions. The database ensures that actions do not interfere with one another during runtime. Effectively, all actions can be considered to execute within atomic blocks, and hence, so are the loops we are verifying. This gives us to freedom to model operations within a loop in any order, or no specific order at all as is the case with coexecution, as long as the final effects of this alternative loop execution are identical to the effects of sequential execution.

```
1  class Class {
2    0+ RelatedClass related_objects
3  }
4  class RelatedClass {
5    0+ Class origin_objects inverseof related_objects
6  }
7
8  action delete_related(0+ Class objects) {
9    foreach o: objects {
10     delete o.related_objects
11   }
12 }
```

Figure 5.4: Unparallelizable but coexecutable loop

5.1.1 Coexecution vs Parallel Execution

Coexecution intuitively resembles parallel execution, but is fundamentally different. First and foremost, concurrency is well known to be more difficult to verify than single threaded execution [35]. Modeling concurrent execution of iterations would make verification even less feasible, which is contrary to our goal. To illustrate the difference between parallel execution and coexecution, we present a loop with coexecutable iterations that are not parallelizable in Figure 5.4 and a loop with parallelizable iterations that are not coexecutable in Figure 5.5.

The loop in Figure 5.4 demonstrates how exclusivity of domains touched by iterations (the condition for parallelization) is too strict a condition for our purpose. This data store has two classes `Class` and `RelatedClass`, whose objects can be associated with many-to-many cardinality. The action takes a set of any number of `Class` objects as an argument `objects` (line 8), iterates over this set and deletes all associated objects (lines 9-11). We observe that this loop could be coexecuted, implying simultaneous deletion of all `RelatedClass` objects that are related to at least one object in the `objects` set. However, these iterations are not parallelizable because multiple iterations may attempt to delete the same objects (in case any two distinct objects in `objects` are associated with the same `RelatedClass` object).

```
1  class Class1 {}
2  class Class2 {}
3
4  action delete_class2_if_no_class1 {
5    foreach ... {
6      if not isempty(allof(Class1)) {
7        delete allof(Class1)
8      } else {
9        delete allof(Class2)
10     }
11   }
12 }
```

Figure 5.5: Parallelizable but not coexecutable loop

It is a well known result that, if operations are atomic and commutative, then they are also parallelizable [91]. Figure 5.5 demonstrates a loop whose iterations are commutative but whose coexecution could yield a different result then any sequential execution.

The set of objects iterated upon is irrelevant for the purposes of this demonstration (line 5), and let us assume the iteration will be executed more than one time. Each iteration tests whether objects of class `Class1` exist or not (line 6). If they exist, they are deleted in line 7. Otherwise, all objects of class `Class2` are deleted (line 9).

These iterations are commutative, as the same result is achieved for any order of iterations. For all sequential executions of more than one iteration, this loop will delete all objects of classes `Class1` and `Class2`.

However, coexecution of these iterations produces a different result. If we were to execute all iterations in isolation from a pre-state that contains an object of class `Class1`, then all isolated iteration executions delete the objects of `Class1`. Combining these isolated operations into one operation produces the same operation as they include nothing but deleting objects of `Class1`, and applying this combined operation on the pre-state leaves objects of `Class2` existing.

5.2 Formalization

Here we formally define coexecution. We also give a condition under which coexecution is equivalent to sequential execution, and we call this property coexecutability. Note that multiple iterations of a loop correspond to repeated sequential execution of the loop body.

In order to simplify our presentation, we will discuss how any two statements A and B can be coexecuted (which, for example, can represent the execution of the same loop body twice for different values of the iterator variable). This discussion can be extended to coexecution of any number of statements, and, hence, is directly applicable to loops by treating iterations of a loop as separate statements.

For brevity and simplicity, we will assume that the data store we reason about contains only one class called `Class` and only one association called `association` that associates objects of type `Class` with objects of type `Class` with many-to-many cardinality. This allows us to use minimal notation for data-store states, avoiding the need to explicitly provide type information. For example, the state $\{a, b, c, \langle a, b\rangle, \langle a, c\rangle\}$ contains exactly three objects of type `Class`, as well as two tuples of the `association` type that associate object a with the other two.

As we discussed earlier, given two statements A and B, their sequential composition $A; B$ is defined by the sequential execution formula below:

$$\forall s, s' \in \overline{DS} \times \overline{DS} : [s, s']_{A;B} \Leftrightarrow \exists s_i : [s, s_i]_A \wedge [s_i, s']_B \tag{1}$$

5.2.1 Execution Deltas

In order to define coexecution, we first need to define a way to express the effects of executing statements. Without loss of generality, assume that the set of role assignments

is immutable. Let us define a structure $\langle O_c, T_c, O_d, T_d \rangle$ for that purpose: O_c and T_c are sets of objects and tuples, respectively, that are created by a given execution, and O_d and T_d are sets of objects and tuples, respectively, that are deleted by a given execution. Let us call this structure the *delta* of an execution.

Given an execution from state s to s', we denote the delta of this execution as $s' \ominus s$. For example, if given two states $s_1 = \{a, b, \langle a, b \rangle\}$ and $s_2 = \{a, b, c\}$, then $s_2 \ominus s_1 = \langle \{c\}, \{\}, \{\}, \{\langle a, b \rangle\} \rangle$ and $s_1 \ominus s_2 = \langle \{\}, \{c\}, \{\langle a, b \rangle\}, \{\} \rangle$.

A delta is *consistent* if and only if its corresponding create and delete sets are mutually exclusive, i.e.,

$$
\begin{aligned}
& O_c \cap O_d = T_c \cap T_d = \emptyset \wedge \\
& (\forall t = \langle o_o, o_t \rangle \in T_c : o_o \notin O_d \wedge o_t \notin O_d) \wedge \\
& (\forall t = \langle o_o, o_t \rangle \in T_d : o_o \notin O_c \wedge o_t \notin O_c)
\end{aligned}
$$

In order to combine the changes done by different executions, we introduce the union ($\cup$) of two deltas:

$$
\begin{aligned}
& \forall \delta_1 = \langle O_{c1}, T_{c1}, O_{d1}, T_{d1} \rangle, \delta_2 = \langle O_{c2}, T_{c2}, O_{d2}, T_{d2} \rangle : \\
& \delta_1 \cup \delta_2 = \langle O_{c1} \cup O_{c2}, T_{c1} \cup T_{c2}, O_{d1} \cup O_{d2}, T_{d1} \cup T_{d2} \rangle
\end{aligned}
$$

We will use this operation to merge the changes done by independently executed statements. Note that the result of the union operation may not be a consistent delta even if all the arguments were individually consistent. We call deltas *conflicting* if and only if their union is not consistent.

5.2.2 Delta Apply Operation

We will introduce the *apply* operation, that takes a state s and a consistent delta δ and updates the state as dictated by the delta. The result is a new state that contains all objects and tuples that existed in s and were not deleted by δ, and all objects and tuples created by δ. In addition, whenever an object is deleted, all the tuples referring to that object are deleted as well. The apply operation maps a state and a consistent delta into a state, and we use the $\oplus$ operator to denote this operation. Formally, given a state s and a consistent delta $\delta = \langle O_c, T_c, O_d, T_d \rangle$:

$$
\begin{aligned}
&\forall s = \langle O, T, U \rangle \in \overline{DS}, s' = \langle O', T', U' \rangle \in \overline{DS} \colon s' = s \oplus \delta \Leftrightarrow \\
&\quad (\forall o \colon o \in O' \Leftrightarrow (o \in O \vee o \in O_c) \wedge o \notin O_d) \wedge \\
&\quad (\forall t = \langle l, o_o, o_t \rangle \colon t \in T' \Leftrightarrow (t \in T \vee t \in T_c) \wedge t \notin T_d \wedge o_o \in O' \wedge o_t \in O') \wedge \\
&\quad U = U'
\end{aligned}
$$

For example, given a state $s = \{a, b, c, \langle a, b \rangle\}$ and a delta $\delta = \langle \{c\}, \{b\}, \{\langle a, c \rangle\}, \{\} \rangle$, $s \oplus \delta = \{a, c, \langle a, c \rangle\}$. Notice how the creation of object c was idempotent given that s already had that object, and that deletion of object b implied that all tuples related to b were deleted as well.

We can observe that $\forall s, s' \in \overline{DS} \times \overline{DS} : s' = s \oplus (s' \ominus s)$. This follows directly from the definition, as $s' \ominus s$ will create all entities (objects and tuples) in s' that are not in s and delete all the entities that are part of s and not s'.

Lemma 1 *Given any two non-conflicting deltas δ_1 and δ_2:*

$$\forall s \in \overline{DS} : (s \oplus \delta_1) \oplus \delta_2 = s \oplus (\delta_1 \cup \delta_2)$$

This lemma follows directly from definitions of delta union and the apply operation. For simplicity we will limit the proof to objects, but the same proof can be extended to cover tuples.

Given a state $s = \langle O, T, U \rangle$, non-conflicting deltas $\delta_1 = \langle O_{c1}, T_{c1}, O_{d1}, T_{d1} \rangle$ and $\delta_2 = \langle O_{c2}, T_{c2}, O_{d2}, T_{d2} \rangle$, and post-states $s_s = \langle O_s, T_s, U_s \rangle = (s \oplus \delta_1) \oplus \delta_2$ and $s_p = \langle O_p, T_p, U_p \rangle = s \oplus (\delta_1 \cup \delta_2)$, we proceed to show that any object in s_s must be in s_p, and that any object in s_p must be in s_s.

$$\begin{aligned} & \forall o \in O_s : (o \in O \wedge o \notin O_{d1} \wedge o \notin O_{d2}) \vee \\ & \qquad (o \in O_{c1} \wedge o \notin O_{d2}) \vee o \in Oc_2 \\ \Rightarrow & \forall o \in O_s : (o \in O \wedge o \notin O_{d1} \wedge o \notin O_{d2}) \vee \\ & \qquad (o \in O_{c1} \vee o \in Oc_2) \end{aligned}$$

Because these deltas are non-conflicting, $(o \in O_{c1} \vee o \in O_{c2}) \Rightarrow (o \notin O_{d1} \wedge o \notin O_{d2})$. Joining this implication with the previous formula:

$$\begin{aligned} & \forall o \in O_s : (o \in O \wedge o \notin O_{d1} \wedge o \notin O_{d2}) \vee \\ & \qquad ((o \in O_{c1} \vee o \in O_{c2}) \wedge o \notin O_{d1} \wedge o \notin O_{d2}) \\ \Rightarrow & \forall o \in O_s : (o \in O \vee o \in O_{c1} \vee o \in O_{c2}) \wedge o \notin O_{d1} \wedge o \notin O_{d2} \\ \Rightarrow & \forall o \in O_s : (o \in O \vee o \in O_{c1} \cup O_{c2}) \wedge o \notin O_{d1} \cup O_{d2} \\ \Rightarrow & \forall o \in O_s : o \in O_p \end{aligned}$$

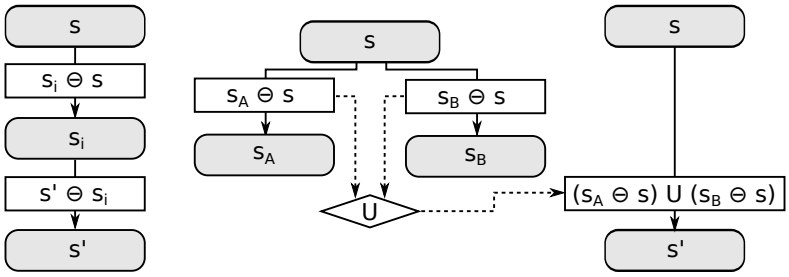


Figure 5.6: Sequential execution vs. coexecution

The inverse implication also holds:

$$\begin{aligned}
&\forall o \in O_p : (o \in O \vee o \in O_{c1} \cup O_{c2}) \wedge o \notin O_{d1} \cup O_{d2} \\
\Rightarrow &\forall o \in O_p : (o \in O \wedge o \notin O_{d1} \wedge o \notin O_{d2}) \vee \\
&\qquad (o \in O_{c1} \wedge o \notin O_{d2}) \vee o \in O_{c2} \\
\Rightarrow &\forall o \in O_p : o \in O_s
\end{aligned}$$

A consequence of this property is that the delta apply operation is commutative for non-conflicting deltas (as delta union is trivially commutative).

5.2.3 Coexecution

Coexecution of two statements A and B, which we denote as $A|B$, means finding the deltas of independent executions of both statements starting from the pre-state, finding the union of those deltas, and applying the union to the pre-state. This is visualized in Figure 5.6, similar to Figures 5.2 and Figures 5.3 but applied to two generic states and any two statements A and B. Formally,

$$\begin{aligned}
&\forall s, s' \in \overline{DS} \times \overline{DS} : [s, s']_{A|B} \Leftrightarrow \exists s_A, s_B \in \overline{DS} \times \overline{DS} : \\
&[s, s_A]_A \wedge [s, s_B]_B \wedge s' = s \oplus ((s_A \ominus s) \cup (s_B \ominus s)) \qquad (2)
\end{aligned}$$

Notice that coexecution, because of the delta apply operation, is defined only if no two possible deltas from the pre-state via statements A and B are conflicting.

For example, if statement A adds a new object to a state, and statement B deletes all tuples from a state, executing these statements from the state $s = \{a, b, \langle a, b\rangle\}$ independently will yield the following states:

$$s_A = \{a, b, c, \langle a, b\rangle\}, \quad s_B = \{a, b\}$$

Therefore

$$\begin{aligned} s_A \ominus s =& \langle\{c\}, \{\}, \{\}, \{\}\rangle \\ s_B \ominus s =& \langle\{\}, \{\}, \{\}, \{\langle a, b\rangle\}\rangle \\ (s_A \ominus s) \cup (s_B \ominus s) =& \langle\{c\}, \{\}, \{\}, \{\langle a, b\rangle\}\rangle \end{aligned}$$

and, the coexecution $A|B$ will result in the following state:

$$s \oplus ((s_A \ominus s) \cup (s_B \ominus s)) = \{a, b, c\}$$

which is the same state to which sequential execution of A and B would transition from s, which means that A and B are coexecutable.

5.2.4 Coexecutability Condition

Not all statements are coexecutable since coexecution requires non-conflicting deltas, and even if their deltas are not conflicting, the result of coexecution may not be equal to the result of sequential execution. Below we define a *coexecutability condition*, such that, given any two statements A and B, if A and B satisfy the coexecutability condition, then

their sequential execution is always equivalent to their coexecution.

We have already shown that, given two non-conflicting deltas, applying them sequentially is equivalent to applying their union. Therefore, $[s, s']_{A|B}$ if and only if $\exists s_A, s_B \in \overline{DS} \times \overline{DS} : [s, s_A]_A \wedge [s, s_B]_B \wedge s' = (s \oplus \delta(s_A \ominus s)) \oplus \delta(s_B \ominus s)$. This view of coexecution gives us a candidate for the coexecutability condition: If the set of deltas achievable from s via B is equal to the set of deltas achievable from s_i via B for any s_i where $[s, s_i]_A$, coexecution and sequential execution are equivalent.

This condition, while valid, is too restrictive. For example, let us consider the statement that deletes all objects of a particular class. Let both A and B be this statement, corresponding to the case of executing this statement twice in a row. Since both A and B always transition from s to an empty state, there is only one delta achievable from s and it will delete all contents of s. When we look at the sequential execution formula (1), the only state s_i such that $[s, s_i]_A$ is an empty state $s_i = \emptyset$. Subsequently, the only delta achievable from an empty state s_i via B is an empty delta. Hence, the deltas achievable from s_i via B are different from the deltas achievable from s via B (assuming s is not itself empty). If we were to use the coexecutability condition mentioned above, we would conclude that these two statements are not coexecutable. However, these two statements are indeed coexecutable. During coexecution both deltas from s via A and B will delete all contents of s, the union of these deltas will do the same, and the result of applying this delta union to s will yield an empty post-state which is equivalent to the result of sequential execution.

The reason why these delete-all statements A and B are coexecutable is that the only thing preventing B from achieving the same deltas from s and s_i is that A does part of B's work while transitioning from s to s_i. B's intent to delete all objects is not altered by A's operations. We will now proceed to define a condition under which a statement's behavior is affected by execution of another statement.

Statement Reads, Creates and Deletes

We model each statement as a set of (potentially non-deterministic) state transitions. This definition of statements is very general and widely applicable, but makes it difficult to identify a statement's read set. We need to have access to a statement's read set in order to reason about interdependencies of statements. In the remainder of this subsection we define how to infer a statement's read, create and delete sets from its transition set.

Note that these definitions are different from the definitions we introduced for the purpose of access control verification in Chapter 4. More specifically, before by *read* we refer only to data that is directly exposed to the user via the view. Here we refer to a more general operation of loading data from the database inside a single statement, regardless of the purpose.

First, we define what it means for a delta set Δ to *cover* a given statement A from a given set of states $S = \{s_1, s_2 \dots s_n\}$:

$$\begin{aligned}\text{cover}(\Delta, A, S) \Leftrightarrow &(\forall s \in S, s' \in \overline{DS} : [s, s']_A \Rightarrow (\exists \delta \in \Delta : s' = s \oplus \delta)) \\ &\wedge \ (\forall s \in S, \delta \in \Delta : [s, s \oplus \delta]_A)\end{aligned}$$

I.e., a set of deltas Δ covers a statement A from a set of states S if and only if every state transition achievable from any state in S via A is achievable from the same state via some delta in Δ, and any transition achievable from any $s \in S$ via any delta in Δ is a transition of A.

A delta cover precisely describes all possible executions of a statement from a set of states using a single set of deltas. Intuitively, the existence of a delta cover shows that the given statement does not need to distinguish between the covered states in order to

decide how to proceed with execution.

Note that this does not mean that Δ is the collection of all deltas achievable from states in S via A. We can demonstrate this by considering a delete-all statement with $S = \overline{DS}$, and Δ containing a single delta that creates no entities and deletes all entities that exist in any state in $\overline{DS}$. In this particular case, the delta in Δ is different than any delta achievable from any finite $s \in S$ via A, yet it is true that this Δ covers A from S.

We can now define what it means for a statement A to *read* an entity e:

$$\text{reads}(A, e) \Leftrightarrow \exists s \in \overline{DS} : \neg\exists \Delta \subseteq \overline{DS}_{\Delta} : \text{cover}(\Delta, A, \{s \cup \{e\}, s \setminus \{e\}\})$$

This means that A reads e if and only if there exists a pair of states $s \cup \{e\}$ and $s \setminus \{e\}$ that cannot be covered by any Δ for the statement A. This implies that A's actions are dependent on e's existence in some way, for example if it is deciding whether to delete or not delete some object other than e based on e's existence. I.e., if statement A reads entity e, then in order to describe the behavior of A, we need to specifically refer to e.

Based on this definition, a delete-all statement does not read any entity e because, for any two states $s \cup \{e\}$ and $s \setminus \{e\}$ for any state s, there exists a Δ that covers it: $\Delta = \{\langle\{\}, \{\}, \text{objects of } (s \cup \{e\}), \text{tuples of } (s \cup \{e\})\rangle\}$. Hence, using this definition, we are able to infer that two delete-all statements that are executed back to back are coexecutable, although in sequential execution, behavior of the second delete-all statement changes (it becomes a no-op) due to the presence of the first delete-all statement.

We can also define what it means for a statement A to *create* or *delete* an entity similarly:

$$\text{creates}(A, e) \Leftrightarrow \exists s, s' \in \overline{DS} : [s, s']_A \wedge e \notin s \wedge e \in s'$$

$$\text{deletes}(A, e) \Leftrightarrow \exists s, s' \in \overline{DS} : [s, s']_A \wedge e \in s \wedge e \notin s'$$

Recall that, since we are abstracting away the basic types, any update to the data store state consists of creation and deletion of entities (i.e., objects and associations).

Coexecutability Condition Definition and Proof

We can now define the coexecutability condition and our main result:

Theorem 1 *Given two statements A and B, if the following condition holds:*

$$\forall s \in \overline{DS}, \forall e \in s : (\mathit{reads}(A, e) \Rightarrow \neg \mathit{creates}(B, e) \wedge \neg \mathit{deletes}(B, e))$$
$$\wedge (\mathit{creates}(A, e) \Rightarrow \neg \mathit{reads}(B, e) \wedge \neg \mathit{deletes}(B, e))$$
$$\wedge (\mathit{deletes}(A, e) \Rightarrow \neg \mathit{reads}(B, e) \wedge \neg \mathit{creates}(B, e))$$

then coexecution of A and B is equivalent to their sequential execution (i.e., $\forall s, s' \in \overline{DS}$: $[s, s']_{A;B} \Leftrightarrow [s, s']_{A|B}$). In other words, A and B are coexecutable.

The proof of the above theorem is tedious due to the differences between objects and tuples and how they depend on one another (e.g., deleting an object deletes all associated tuples, and creating a tuple that is associated with a non-existing object is impossible etc.). In order to simplify the proof, without loss of generality, we will outline the proof by focusing only on the creation and deletion of objects.

First, the condition in Theorem 1 implies that no statement can delete an object that can be created by the other. Therefore the deltas from any s via A and B are not conflicting, and coexecution is always defined.

Let us take any two states s and s' and assume that there exists a state s_A such that $[s, s_A]_A$.

Let us consider any object o_c that is created by $s_A \ominus s$. All objects created by $s_A \ominus s$ are not read by B. Therefore, there exists a delta cover Δ that describes all transitions

from $s \cup \{o_c\}$ and $s \setminus \{o_c\}$ via B. Since $s_A \ominus s$ is creating o_c we know that $o_c \notin s$, so this delta cover describes all transitions from s and $s \cup \{o_c\}$ via B.

Let us inspect every member of such a delta set Δ. If any $\delta \in \Delta$ creates anything in s then this operation is always redundant for states s and $s \cup \{o_c\}$, so we can remove this operation and still have a delta cover of B over $\{s, s \cup \{o_c\}\}$. We can similarly remove all deletions of all objects outside $s \cup \{o_c\}$ as redundant operations. Since o_c cannot be deleted by B, we know that this trimmed delta cover does not delete anything outside s. From the definition of a delta cover it follows that the resulting trimmed delta cover is, in fact, precisely the set of all deltas achievable from s via B.

Similar reasoning can be followed for any object o_d that is deleted by $s_A \ominus s$. It follows that the set of deltas achievable from s via B covers B over $\{s, s \setminus \{o_d\}\}$.

Because the set of all deltas achievable from s via B covers $\{s, s \cup \{o_c\}\}$, directly from the definition of delta covers (with the prior assumption that an s_A s.t. $[s, s_A]_A$ exists):

$$\forall s, s' \in \overline{DS} \times \overline{DS} : \exists s_A : [s, s_A]_A \Rightarrow$$
$$([s \cup \{o_c\}, s']_B \Leftrightarrow \exists s_B \in \overline{DS} : [s, s_B]_B \wedge s' = s_B \cup \{o_c\})$$

Let us generalize and say that $s_A \ominus s$ creates objects o_{ci} for some $1 \leq i \leq n_c$ and deletes objects o_{di} for some $1 \leq i \leq n_d$. If we were to now enumerate all objects created and deleted by $s_A \ominus s$ one by one and apply the above reasoning to them, the resulting formula would be:

$$\forall s, s' \in \overline{DS} \times \overline{DS} : \exists s_A \in \overline{DS} : [s, s_A]_A \Rightarrow$$
$$([s \cup \{o_{c1}, \ldots o_{cn_c}\} \setminus \{o_{d1}, \ldots o_{dn_d}\}, s']_B \Leftrightarrow \exists s_B \in \overline{DS} :$$
$$[s, s_B]_B \wedge s' = s_B \cup \{o_{c1}, \ldots o_{cn_c}\} \setminus \{o_{d1}, \ldots o_{dn_d}\})$$

Which is equivalent to

$$\forall s, s' \in \overline{DS} \times \overline{DS} : \exists s_A \in \overline{DS} : [s, s_A]_A \Rightarrow$$
$$[s \oplus (s_A \ominus s), s']_B \Leftrightarrow \exists s_B \in \overline{DS} : [s, s_B]_B \wedge s' = s_B \oplus (s_A \ominus s)$$

Because $s_B = s \oplus (s_B \ominus s)$, and applying non-conflicting deltas in sequence is equivalent to applying their union, this formula is equivalent to

$$\forall s, s' \in \overline{DS} \times \overline{DS} : \exists s_A \in \overline{DS} : [s, s_A]_A \Rightarrow$$
$$([s_A, s']_B \Leftrightarrow \exists s_B \in \overline{DS} : [s, s_B]_B \wedge s' = s \oplus (s_B \ominus s) \cup (s_A \ominus s))$$

We can move the s_A quantification and implication ($\exists s_A \in \overline{DS} : [s, s_A]_A \Rightarrow \dots$) to both sides of the inside equivalence:

$$\forall s, s' \in \overline{DS} \times \overline{DS} :$$
$$(\exists s_A \in \overline{DS} : [s, s_A]_A \wedge [s_A, s']_B) \Leftrightarrow$$
$$(\exists s_A, s_B \in \overline{DS} \times \overline{DS} : [s, s_A]_A \wedge [s, s_B]_B$$
$$\wedge s' = s \oplus (s_B \ominus s) \cup (s_A \ominus s))$$

which is the formula for equivalence of sequential execution and coexecution.

5.3 Syntactic Analysis

In order to keep our verification process fully automatic, we developed a syntactic check that determines, for a given `ForEach` loop, whether we can coexecute the iterations while maintaining the loop semantics. Our syntactic analysis works on an intermediate

abstract data store (ADS) language defined in Chapter 2.4.

The syntactic check is two-fold: 1) we analyze if sequential execution is necessary to uphold variable dependencies, and 2) if iteration operations may overlap as defined in the coexecutability condition (Theorem 1).

First, to check if coexecution would invalidate variable dependences, we convert the whole action to static single assignment (SSA) form. If, after converting to SSA and removing unnecessary assignments, there exists a Phi function assignment at the beginning of the loop's iteration body (i.e. an iteration reads a variable assigned to by a previous iteration) or at the end of the loop (i.e. the iteration assigns to a variable that is read after the loop terminates), then iterations must be modeled sequentially to preserve variable state.

If the variable dependency check passes, we proceed to check whether the loop is coexecutable. To achieve this, we identify every data store class or association that is touched by a read, create or delete operation inside the iteration body.

For example, if a `Delete` statement deletes a set of objects of class c, we mark that c as well as all c's subclasses have had a delete operation executed. In addition, since all tuples of deleted objects are deleted as well, we mark all associations of these classes and their supertypes as having had a delete operation executed.

We increase the precision of our analysis by identifying whether operations are executed on iteration-local objects. For example, if an iteration were to create an object of class c and subsequently delete it, then the coexecutability condition would not be violated (since no object created by one iteration would be deleted by another) but the above syntactic check would fail as c would have had both a create and a delete operation executed.

In order to facilitate this we denote whether each read, create or delete operation is done iteration-locally or not. For example, `CreateObject` creates an object iteration-locally

```
1  program Analysis
2  var data: AnalysisData;
3
4  function Analyze(loop: ForEach): Boolean
5    data.clearAllData;
6    AnalyzeStatement(loop);
7    for type in DataStoreTypes:
8      operations = data.operationsDoneOn(type);
9      if (operations.hasTwoDifferentOpsWithOneGlobal()) then
10       return False;
11     end;
12   end;
13   return True;
14 end
15
16 procedure AnalyzeStatement(stmt: Statement)
17   case type(stmt) of
18   Block:
19     for subStmt in stmt.subStatements do
20       AnalyzeStatement(subStmt);
21     end;
22   Delete:
23     <objSetType, objSetLocal> = AnalyzeObjset(stmt.objSet)
24     data.markDelete(objSetType, objSetLocal);
25     for relation in objSetType.associations do
26       data.markDelete(relation, False);
27     end;
28   ObjectSet:
29     AnalyzeObjset(stmt.objSet);
30   Assign:
31     <objSetType, objSetLocal> = AnalyzeObjset(stmt.objSet);
32     stmt.variable.objSetType = objSetType;
33     stmt.variable.objSetLocal = objSetLocal;
34   ForEach:
35     <objSetType, objSetLocal> = AnalyzeObjset(stmt.objSet);
36     data.markRead(objSetType, objSetLocal);
37     stmt.iteratorVariable.objSetType = objSetType;
38     stmt.iteratorVariable.objSetLocal = objSetLocal;
39     AnalyzeStatement(stmt.block, data);
40   ...
41   end;
42 end
43
44 function AnalyzeObjset(objSet: ObjectSet): <Type, Boolean>
45   case type(objSet) of
46   CreateObject:
47     data.markCreate(objSet.createdType, True);
48     return <objSet.createdType, True>;
49   Variable:
50     return <objSet.objSetType, objSet.objSetLocal>;
51   Dereference:
52     <originType, originLocal> =
53         AnalyzeObjset(objSet.originObjSet);
54     data.markRead(originType, originLocal);
55     data.markRead(objSet.relation, False);
56     return <objSet.targetType, False>;
57   ...
58   end
59 end
```

Figure 5.7: Syntactic analysis pseudocode

as every iteration will create a different object and these created sets will not overlap. Operations such as dereferencing from an object set return a global domain object set even if a local domain was dereferenced, because even if each iteration dereferences from a different object domain, the target object sets may overlap.

Therefore, in order for the syntactic check to pass, there must not exist a domain of objects or tuples that has two of the operations (read, create, delete) executed, where at least one of this operations is not done iteration-locally. The pseudocode for the operation domain analysis is provided in Figure 5.7.

The `AnalysisData` global variable called `data` (line 2) aggregates information about which domains of objects and tuples are operated on and in what way. It is essentially a key-value structure that maps every class and association in the data store into a set of *operation entries*, which are pairs $\langle o, l \rangle$ where $o \in \{create, read, delete\}$ and $l \in \{True, False\}$. This structure lists, for each data store class and association, all the different `create`, `read` and `write` operations executed on entities of that particular class or association and if these operations were executed iteration-locally (`True`) or not (`False`). This structure is populated by invoking methods `markRead`, `markCreate` and `markDelete` on it, all of which take two arguments: a data store class or association type, and a boolean denoting whether the operation is iteration-local (e.g. line 24).

The `Analyze` function is the entry point of our algorithm (lines 4-14). It first clears all information from the `data` object (line 5), then proceeds to gather information in the `data` object by invoking the `AnalyzeStatement` on the given loop (line 6). It then iterates over all classes and associations (lines 7-12) and tests whether there exist two operation entries on the same class or association such that they contain different $\{create, read, delete\}$ types where at least one of them is executed on a global domain. If such a pair of operation entries is found, the coexecutability check fails and the function returns `False` (line 10). Otherwise, it returns `True` (line 13).

The `AnalyzeStatement` procedure takes a statement as an argument and its purpose is to populate the `data` object with information about which operations are executed on which domain by that statement. For example, a `Delete` statement (lines 22-27) invokes the `AnalyzeObjset` method to acquire domain information about the object set to be deleted (line 23), then marks this domain as deleted (line 24). Since the `Delete` statement also deletes all tuples of the deleted objects, all associations around the object set's type are iterated over (lines 25-27) and are marked deleted globally (line 26). Tuples are always deleted on a global domain because, even if the deletion is on a local domain, these tuples may relate to some other iteration's local domain.

The `Assign` statement (lines 30-33) does not add any entry to the `data object`, and instead stores the domain of the assigned object set in the variable. These values will later be extracted when the variable is referred to in lines 49-50.

The `AnalyzeObjset` function (lines 44-59) is invoked with an object set argument and it returns the domain of the objects inside the object set in form of a $\langle Type, Boolean \rangle$ pair. In addition, object sets may populate the `data` object themselves. For example, the `CreateObject` object set (lines 46-49) creates a new object and returns a singleton set containing it. For each such object set, we mark that object's class with the create operation (line 47) in an iteration-local domain because this object set will contain a different object for each iteration.

The `Dereference` object set (lines 51-56) takes another object set, referred to as the origin object set, and an association type. It contains all the objects that can be reached from the origin object set via at least one tuple of the given association type. As such, the origin object set is read in the domain supplied by it (lines 52-54), and the association type is read on the global domain (line 54). Finally, the returned domain of this very `Dereference` object set is equal to the target type of the association and is always global (line 56).

5.4 Experimental Evaluation

In order to evaluate the effect of coexecution on the verification process, we implemented two ways to model `ForEach` loops (as sequentially executed iterations, and as coexecuted iterations). These experiments were originally run using an earlier version of our tool that does not support many-sorted logic (presented in Chapter 6), and as such, uses only Spass for theorem proving. After introducing many-sorted logic and the corresponding translation that enables us to use Z3, we compared sequential execution and coexecution on our larger dataset in Section 6.4.4. Our findings can only be explained after the discussions present in Chapter 6.

These experiments were ran on a sample of actions with loops from some of the most popular (most starred) Rails applications hosted on Github. We found a total of 38 loops in actions of these applications: 5 in Discourse [30], 9 in FatFreeCRM [36], 5 in Tracks [103], 4 in Lobsters [72], 5 in SprintApp [95], 8 in Redmine [89], and 1 in Kandan [64]. The loops we extracted contain various program structures such as branches, object and association creation and deletion as well as loop nesting. Our analysis determined that all these loops were coexecutable. In these experiments we stopped verification after 5 minutes, at which point we deemed the result as inconclusive.

Interestingly, 12 of the 38 loops we extracted had empty loop bodies. This is due to the fact that the abstract data store model we extract abstracts away the fields with basic types. Hence, loops that do not modify the state of the data store as far as the set of objects and associations are concerned (but might change the value of basic type fields of some objects) result in empty loop bodies. Note that loops with empty bodies trivially preserve data integrity and correctly enforce access control. However, during our experiments, we found out that Spass would occasionally timeout even for loops with an empty body when the sequential semantics is used. This demonstrates the inherent

complexity of reasoning about the sequential loop model, even without the complexity of reasoning about the statements in the loop body.

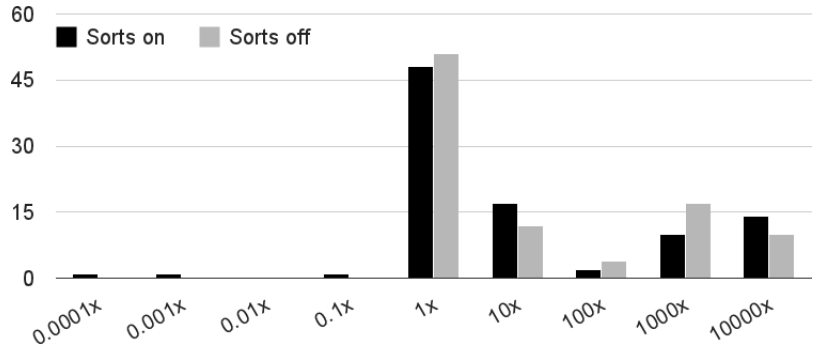
Spass guides the formula space exploration using heuristics that can be fine tuned by the user. We used two heuristics: one with the *Sorts* option on, and the other with that option off. The Sorts option allows Spass to make decisions based on soft sorts [106]. By turning the Sorts option on and off and looking at the deduction logs of Spass we noticed that the order of deduction Spass takes changes significantly, so in theory, one of these heuristics may terminate when the other one does not. Therefore, running Spass with the Sorts option on and off gives us very different heuristics for comparison of coexecution and sequential execution.

Normally, when we encode an action/invariant pair or an authorization property in FOL, loop semantics are encoded as axioms with an invariant or authorization property being the conjecture. In order to isolate the effect of the axioms on the overall deduction process, we also verified all actions using the conjecture *false*. This conjecture often gives us the worst case performance for a set of axioms. Because Spass attempts to deduce a contradiction from the axioms and negated conjecture, it negates the conjecture to *true* and hence needs to explore the entire space of deducible formulae to reach a contradiction that does not exist. If Spass terminates with the *false* conjecture then we can reasonably expect that it will terminate with other invariants as long as they do not add significant complexity to the verified theorem. We also included 4 actions that we manually created to explore how the theorem prover handles coexecution vs sequential execution of nested loops and branches in iterations.

In total, we had 94 action/conjecture pairs. We translated each one of those to two FOL theorems, one using coexecution and the other using sequential execution to model loops. We sent each one of these theorems to two instances of Spass with different heuristic settings, resulting in 376 verification tasks. These verification experiments were

Loop Model	Heuristic	# of Timeouts / Total	Avg Time (seconds)
Sequential	Sorts on	66/94 (70.2%)	216.4
	Sorts off	56/94 (60.2%)	186.0
Coexecution	Sorts on	27/94 (29.3%)	94.0
	Sorts off	18/94 (19.8%)	68.5

(a) Verification results



(b) Number of action/invariants per performance gain factor

Figure 5.8: Application information and verification results

executed on a computer with an Intel Core i5-2400S processor and 32GB RAM, running 64bit Linux. Memory consumption never exceeded 200Mb.

We specifically looked at how coexecution fared as opposed to sequential execution. With the sequential execution model, out of 188 verification tasks, 122 timed out (64.89%). With the coexecution model, only 45 tasks out of 188 timed out (24.19%). The summary of our results can be seen on Figure 5.8. Figure 5.8(a) summarizes the number of timeouts and average verification time over loop interpretation (sequential vs coexecution) and heuristic.

Figure 5.8(b) summarizes the performance effects of coexecution as opposed to sequential execution. We had 188 cases in which to compare coexecution and sequential execution under identical action/invariants and theorem prover heuristics. In 86 cases (45%, columns labeled *10x* and up), coexecution improved verification times by at least an order of magnitude. Among them, in 24 cases (13%) the theorem prover reached a conclusive answer instantly using coexecution and could not deduce a conclusive an-

swer at all with sequential execution (column labeled *10000x*). Coexecution yielded no improvement in a total of 99 tasks (52%, column *1x*). In these cases either both loop models resulted in a timeout or both methods produced results instantly. In three cases, coexecution produced worse results than the sequential model. This is not surprising since, as we mentioned above, the proof search implementation of the theorem prover relies on several heuristics which influence its performance.

In total, we found that coexecution reduced the timeout rate from 65% to 24% (almost threefold), made verification at least an order of magnitude faster 45% of the time, with 13% of cases terminating quickly as opposed to not terminating at all. We conclude that, overall, coexecution allows for significantly faster verification and significantly decreases the chance of verification never terminating.

Chapter 6

Verification via Many-sorted Logic

Minimizing the ratio of inconclusive results is a necessary step for making our approach usable in practice. Inconclusive results force the developer to manually investigate actions and invariants, and since we encounter inconclusive results in the most complex actions, this is a difficult and error prone process.

In order to understand the cause of inconclusive results, we investigated the logs of the theorem prover we used in our previous experiments. We noticed that the theorem prover did an excessive number of deductions solely to reason about the types of quantified variables and objects. Since FOL does not have a notion of type, our FOL translation generates predicates that encode all the type information, and the theorem prover was spending a lot of time reasoning about these predicates.

This seemed unnecessary to us, as in general, inheritance is rarely used in web applications. Out of 25 most starred Ruby on Rails applications on Github only 7 employ inheritance, and on average, only 23% of classes inherit or are inherited from other classes. This means that, if FOL would allow us, we could annotate our formulas with precise type information and a theorem prover might use this information to greatly trim the

space of deductions it makes.

There exists a variant of FOL called many-sorted logic. Many-sorted logic enforces a rigid type system on top of FOL, where all predicates, functions etc have to be annotated with types.

In this chapter we present a translation of our model to many-sorted logic, and encounter and fix a problem regarding empty logic in our many-sorted translation. We then show that that using many-sorted logic drastically increased our verification performance, and furthermore, that sorts themselves are the main factor in this performance increase.

6.1 Many-Sorted Logic

Sometimes it is useful to divide the universe of a structure using types with mutually exclusive domains. This is especially true if the functions and predicates make sense only within a specific domain. For example, if we need a language to express integer and string operations, many-sorted logic makes it easy to express that all elements of the universe are either integers or strings. It also allows us to define that, for example, the `stringLength` function always maps a string element to an integer element, and that the `isPositive` predicate can only accept an integer as its argument.

Types in many-sorted logic are called *sorts*. Many-sorted logic requires us to explicitly declare the types of all function and predicate arguments, function return values and variables. It also gives us the ability to quantify over elements of a given type instead of over the whole universe.

Formally, many-sorted logic is very similar to classical FOL. In addition to everything discussed in Section 4.1, L includes a set of sorts S. Functions and predicates in F and P respectively define the sorts of their arguments, functions define the sort of their return

value, and all variables are associated with a sort from S. We also require all formulas to be well typed (e.g. a predicate can only accept a term as an argument if the term's sort matches the predicate's declaration).

A structure S in many-sorted logic does not contain a single universe U. Instead, it contains a non-empty universe U^s for each sort $s \in S$. For each predicate p of sorts $s_1 \dots s_k$ and arity k, we define P^S as a subset of $U^{s_1} \times \cdots \times U^{S_k}$. The set F^U is defined analogously, and V^U assigns an element of a variable's sort to each variable. Quantification is always done over a specific sort's universe. For clarity, we explicitly declare the sort s of a variable v when quantifying by using the notation $\forall s\, v\colon f$.

Note that many-sorted logic and unsorted logic have equivalent expressive power [22]. Given a set of many-sorted formulas, a similar set of unsorted formulas is equisatisfiable if we introduce predicates used to denote sorts and conjoin the formulas that partition the universe to these sorts. Unsorted logic can be translated to many-sorted logic by introducing a single sort that applies to all language elements.

6.2 Empty Logic

Empty universes are a useful concept for data model verification. In general, a data model state may contain no objects. This is an important consideration for data model verification (e.g. does the application behave properly even if there exist no Users or Accounts?). For this reason it is necessary to consider empty universes as a possibility during verification. As one would expect, data model verification tools, such as Alloy [62], support empty domains. However, empty universes are outside the scope of classical FOL. Even though Spass, the tool we used previously, does not support empty universes, our translation was such that the empty model state was a possibility. This will cease to be the case after optimizations introduced in Section 6.3. However, before we explain this

problem, we must define *empty logic*: FOL that allows an empty universe.

FOL universes are typically defined to be non-empty. Allowing the special case of an empty universe makes definitions more complicated, and invalidates certain inference rules that stop working only in the case of an empty universe (for example, $\phi \vee \exists x \psi$ implies $\exists x(\phi \vee \psi)$ where x is not a free variable in ϕ). The treatment of variables and function return values becomes problematic because terms are expected to always take a value of one element of the universe. This is not possible in empty universes.

Furthermore, the possibility of an empty universes breaks certain fundamental rules about FOL. E.g. $\forall x \colon x \neq x$ is normally an unsatisfiable formula. If we define quantification over an empty universe to be vacuously true (as there does not exist an assignment of the variable that does not satisfy the subformula), this example formula is satisfied by a structure with an empty universe.

Empty logic is a variant of FOL that allows empty universes. The treatment of empty universe in empty logic is defined by Quine [86]: universal quantification over an empty set is considered vacuously true (since there exists no counterexample variable assignment), and existential quantification over an empty set is considered vacuously false (since there exists no satisfactory variable assignment).

This interpretation of quantification over empty sorts is in concordance with an alternative definition of universal quantification: Given a universe U, quantification $\forall v \colon f$ can be unrolled into a conjunction of all formulas that result from replacing v in f with an element of U. In case of an empty universe this list of quantified formulas is empty, and the neutral element of conjunction is the boolean *true*.

In combination with many-sorted logic, empty logic allows a sort's universe to be empty. Although theorem provers we use during verification do not support empty logic, in our translation of data models to FOL, we simulate the empty logic semantics so that the resulting translation covers the data model behaviors where data classes can be

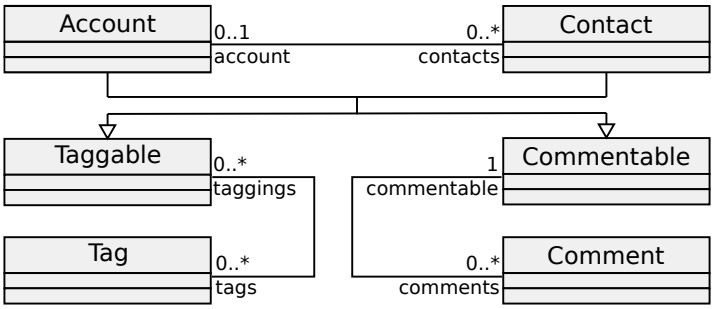


Figure 6.1: A data model schema example based on FatFreeCRM [36]

empty (i.e., without any instances). We discuss our formalization of the data models and how we deal with many-sorted logic and empty universes in our translation to FOL in the following sections.

6.3 Translation to Many-sorted Logic

The translation presented in Chapter 4 is based on unsorted, empty logic. We found [15] that many-sorted logic allows for more efficient theorem proving. In this section we modify the previously presented translation to many-sorted logic. For brevity, we focus only on classes as associations are largely analogous. We will repeat Figures 4.1 and 4.2 in this section as Figures 6.1 and 6.2 for self containment.

Within our translation where universe elements correspond to entities, sorts naturally serve the purpose similar to classes and associations. However, sorts imply disjoint universes, which is only suitable for classes that do not employ inheritance. Classes that employ inheritance cannot be directly mapped to sorts because a subclass's object set is a subset of a parent's.

To work around this problem, we partition the set of all classes into *inheritance clusters*. An inheritance cluster is a maximal set of classes such that, for any two classes c_1 and c_k in the cluster, there exists a list of classes $c_1, c_2, \ldots c_k$ where each consecutive

Predicates: Account, Contact, Taggable, Commentable, Tag, Comment, XTaggable, XCommentable.

$$\forall o\colon \texttt{Account}(o) \rightarrow \texttt{Taggable}(o) \wedge \texttt{Commentable}(o) \quad (1)$$

$$\forall o\colon \texttt{Contact}(o) \rightarrow \texttt{Taggable}(o) \wedge \texttt{Commentable}(o) \quad (2)$$

$$\forall o\colon \texttt{XTaggable}(o) \leftrightarrow \texttt{Taggable}(o) \wedge \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \quad (3)$$

$$\forall o\colon \texttt{XCommentable}(o) \leftrightarrow \texttt{Commentable}(o) \wedge \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \quad (4)$$

$$\forall o\colon \texttt{Account}(o) \rightarrow \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o) \quad (5)$$

$$\forall o\colon \texttt{Contact}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o) \quad (6)$$

$$\forall o\colon \texttt{XTaggable}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o) \quad (7)$$

$$\forall o\colon \texttt{XCommentable}(o) \rightarrow \neg\texttt{Account}(o) \vee \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{Tag}(o) \wedge \neg\texttt{Comment}(o) \quad (8)$$

$$\forall o\colon \texttt{Tag}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Comment}(o) \quad (9)$$

$$\forall o\colon \texttt{Comment}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \wedge \neg\texttt{Tag}(o) \quad (10)$$

Figure 6.2: Axioms defining the class diagram in Figure 6.1 in classical (unsorted) first order logic

pair of classes constitutes a child-parent or parent-child relationship. In other words, in the class graph where vertices are classes and edges correspond to inheritance, an inheritance cluster is a maximally connected component. Note that all classes that do not employ inheritance are members of singleton clusters.

For each inheritance cluster we introduce a sort that is common to all classes in the cluster. In case of an inheritance cluster with multiple classes we introduce predicates and axioms in order to differentiate classes within the cluster. These predicates and axioms are similar in purpose to the predicates used in the unsorted logic translation. For each class c in a non-singleton inheritance cluster we introduce unary predicates $\overline{c}$ and $\overline{c}_x$ of the cluster's sort and introduce axioms that resemble the ones defined for unsorted logic, the key distinction being these axioms refer to classes of that cluster only.

Specifically, inheritance axioms are defined as follows: for each class c that belongs to an inheritance cluster of sort s and whose superclass set is $\text{superclass}(c) = \{p_1, p_2 \ldots p_k\}$:

$$\forall s\, o\colon \overline{c}(o) \rightarrow \overline{p_1}(o) \wedge \overline{p_2}(o) \wedge \cdots \wedge \overline{p_k}(o)$$

For the model presented in Figure 6.1, inheritance axioms are formulas (1) and (2)

Sorts: `Cluster`, `Tag`, `Comment`.
Predicates: `Account`(`Cluster`), `Contact`(`Cluster`), `Taggable`(`Cluster`), `XTaggable`(`Cluster`), `Commentable`(`Cluster`), `XCommentable`(`Cluster`).

$$\forall\ \texttt{Cluster}\ o\colon \texttt{Account}(o) \rightarrow \texttt{Taggable}(o) \wedge \texttt{Commentable}(o) \quad (1)$$
$$\forall\ \texttt{Cluster}\ o\colon \texttt{Contact}(o) \rightarrow \texttt{Taggable}(o) \wedge \texttt{Commentable}(o) \quad (2)$$
$$\forall\ \texttt{Cluster}\ o\colon \texttt{XTaggable}(o) \leftrightarrow \texttt{Taggable}(o) \wedge \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \quad (3)$$
$$\forall\ \texttt{Cluster}\ o\colon \texttt{XCommentable}(o) \leftrightarrow \texttt{Commentable}(o)\ \wedge \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \quad (4)$$
$$\forall\ \texttt{Cluster}\ o\colon \texttt{Account}(o) \rightarrow \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \quad (5)$$
$$\forall\ \texttt{Cluster}\ o\colon \texttt{Contact}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{XTaggable}(o) \wedge \neg\texttt{XCommentable}(o) \quad (6)$$
$$\forall\ \texttt{Cluster}\ o\colon \texttt{XTaggable}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XCommentable}(o) \quad (7)$$
$$\forall\ \texttt{Cluster}\ o\colon \texttt{XCommentable}(o) \rightarrow \neg\texttt{Account}(o) \wedge \neg\texttt{Contact}(o) \wedge \neg\texttt{XTaggable}(o) \quad (8)$$

Figure 6.3: Axioms defining the class diagram in Figure 6.1 in many-sorted logic

in Figure 6.3.

An instance axiom is generated for each class c. Let $\{s_1 \ldots s_k\}$ be the set of c's subclasses and let s be the sort of c's inheritance cluster:

$$\forall s\, o\colon \overline{c}_x(o) \leftrightarrow \overline{c}(o) \wedge \neg\overline{s_1}(o) \wedge \cdots \wedge \neg\overline{s_k}(o)$$

Given the model presented in Figure 6.1, instance axioms are formulas (3) and (4) in Figure 6.3.

Finally, membership axioms are generated for each non-singleton inheritance cluster individually instead of for the entire set C. Given an inheritance cluster that consists of classes $\{c_1, \ldots c_k\}$ where $k > 1$ we generate an axiom for each class c_i inside this cluster:

$$\forall s\, o\colon \overline{c_i}(o) \rightarrow \neg\overline{c_1}(o) \wedge \cdots \wedge \neg\overline{c_{i-1}}(o) \wedge \neg\overline{c_{i+1}}(o) \wedge \cdots \wedge \neg\overline{c_k}(o)$$

Formulas (5)-(8) in Figure 6.3 correspond to membership axioms for the model in Figure 6.1.

The number of introduced predicates and axioms is highly dependent on the data

model in question. With no inheritance, no additional predicates and axioms are introduced. The number and size of formulas introduced by each inheritance cluster are linear in the number of classes in the cluster. However, most classes do not employ inheritance in data models of real world applications (18 out of 25 most starred Ruby on Rails applications do not employ inheritance at all, with an average of 23% classes involving inheritance), making most classes part of singleton inheritance clusters. Furthermore, if multiple non-singleton inheritance clusters exist in the data model, the size of generated axioms is relatively small when compared to those generated by the unsorted logic translation. Finally, in case of a model with only singleton clusters, no additional axioms are required to define the type system.

6.3.1 Empty logic

Our treatment of empty structures is dependent on whether the underlying theory is unsorted or many-sorted. In fact, our translation to unsorted logic as presented in Chapter 4 allows empty structures by default. This becomes clear when we change the interpretation of all type predicates $\overline{c}$ to imply that the universe element in question is of the given type, but in addition, it *exists* semantically. Notice that our encoding does not require that all universe elements are of a class type. For example, we use universe elements to represent tuples, and it is not required for a universe element to represent either an object or a tuple.

Whenever we define functions and predicates in unsorted logic we constrain argument values and the return value, if applicable, to be of expected types. As a corollary of our expanded interpretation, function return values objects exist semantically if and only if arguments exist semantically and are of corresponding types. Similarly, predicates may accept a set of domain elements under the condition that they exist semantically and are

```
1  class CommentsController
2    ...
3    def destroy
4      @comment = Comment.find(params[:id])
5      @comment.destroy
6      respond_with(@comment)
7    end
8    ...
9  end
```

Figure 6.4: Example action based on FatFreeCRM [36]

of corresponding types.

As for quantification, whenever quantifying over a class type, we introduce a condition that the subformula is relevant only for domain elements that represent objects of the given type. For example, in order to universally quantify over elements of class c using the variable v and a subformula f we generate a formula $\forall v\colon \overline{c}(v) \rightarrow f$. In case of existential quantification we would instead generate $\exists v\colon \overline{c}(v) \wedge f$.

Predicates: `PreState`, `PostState`, `AtComment`.

$$\forall x\colon \texttt{AtComment}(x) \Rightarrow \texttt{Comment}(x) \tag{1}$$

$$\forall x\colon (\forall y\colon \texttt{AtComment}(x) \wedge \texttt{AtComment}(y) \Rightarrow x = y) \tag{2}$$

$$\forall x\colon \texttt{AtComment}(x) \Rightarrow \neg\texttt{PostState}(x) \tag{3}$$

$$\forall x\colon \neg\texttt{AtComment}(x) \Rightarrow (\texttt{PreState}(x) \Leftrightarrow \texttt{PostState}(x)) \tag{4}$$

Figure 6.5: Unsorted action translation example

For example, the action presented in Figure 6.4 can be translated to FOL as defined in Figure 6.5. For brevity, we omit listing all predicates and axioms that define the type system. In this translation, the `AtComment` predicate denotes values that are saved in the `@Comment` variable. First we constrain type-specific predicates to refer to their actual types (formula (1)). Note that as part of our interpretation of class type predicates, any entity accepted by the `AtComment` is also accepted by `Comment` and therefore exists semantically. Next, in formula (2) we constrain that there exists at most one element in variable `AtComment`, as the `find` method in Ruby on Rails (line 4 in Figure 6.1) returns at most one object.

Formulas (3) and (4) define the delete statement. Formula (3) defines that the objects in the `@Comment` variable no longer exist after the statement (regardless of their existence before). Formula (4) defines that all objects outside this variable existed before if and only if they exist after the statement has finished executing. This particular translation allows for an empty universe. Such a structure would have no elements accepted by predicates `Comment` and `AtComment`.

The problem with the empty universe becomes more apparent with the many-sorted logic translation. If we were to define a Comment sort and use it alone to define the set of all comments, then the universe of this sort would be non-empty, meaning that at least one Comment would exist for every sort. To go around this problem, for each such class c, we introduce a predicate $\overline{c}$ that accepts a single argument of c's sort. We do not introduce any axioms. We use these predicates to define object sets of these classes, implying that object sets are subsets of their corresponding universes.

Predicates: $\texttt{PreState}_{\texttt{Cluster}}(\texttt{Cluster})$, $\texttt{PreState}_{\texttt{Comment}}(\texttt{Comment})$, $\texttt{PreState}_{\texttt{Tag}}(\texttt{Tag})$, $\texttt{PostState}_{\texttt{Cluster}}(\texttt{Cluster})$, $\texttt{PostState}_{\texttt{Comment}}(\texttt{Comment})$, $\texttt{PostState}_{\texttt{Tag}}(\texttt{Tag})$, $\texttt{AtComment}(\texttt{Comment})$, $\texttt{Comment}_P(\texttt{Comment})$, $\texttt{Tag}_P(\texttt{Tag})$.

$$\forall\ \texttt{Comment}\ x\colon \texttt{AtComment}(x) \Rightarrow \texttt{Comment}_P(x) \tag{1}$$

$$\forall\ \texttt{Comment}\ x\colon (\forall \texttt{Comment} y\colon \texttt{AtComment}(x) \wedge \texttt{AtComment}(y) \Rightarrow x = y) \tag{2}$$

$$\forall\ \texttt{Comment}\ x\colon \texttt{AtComment}(x) \Rightarrow \texttt{PreState}_{\texttt{Comment}}(x) \wedge \neg \texttt{PostState}_{\texttt{Comment}}(x) \tag{3}$$

$$\forall\ \texttt{Comment}\ x\colon \neg\texttt{AtComment}(x) \Rightarrow (\texttt{PreState}_{\texttt{Cluster}}(x) \Leftrightarrow \texttt{PostState}_{\texttt{Cluster}}(x)) \tag{4}$$

Figure 6.6: Many-sorted action translation example

Given the example action in Figure 6.4, a many-sorted translation can be defined as in Figure 6.6. Note that, once again, we omit declaring all sorts, predicates and axioms from Figure 6.3 for brevity.

Notice that we introduce predicates $\texttt{Comment}_P$ and $\texttt{Tag}_P$ in addition to previously defined sorts `Comment` and `Tag`. In Formula (1) we define that all elements accepted by `AtComment` are also accepted by $\texttt{Comment}_P$. This is necessary to express since, with-

out this axiom, there could be an element accepted by `AtComment` that is not accepted by $\texttt{Comment}_P$. Formula (2) defines that there exists at most one element accepted by `AtComment`. Formulas (3) and (4) define how the delete statement transitions between the pre-state and the post-state. These formulas are analogous to formulas (3) and (4) in the unsorted translation. Note that, however, these formulas are constrained to the `Comment` sort. All other sorts are handled implicitly (we do not differentiate between their pre- and post-states). This demonstrates the benefit of introducing sorts, as the theorem prover does not need to reason at all about other types by default.

Empty structures are handled by this translation. For example, a structure that represents this case would have no entities of sort `Comment` be accepted by predicates $\texttt{Comment}_P$ and `AtComment`. Without introducing a predicate $\texttt{Comment}_P$ this would not be the case.

6.4 Experimental Evaluation

We conducted two sets of experiments. Both these experiments involved verification of applications previously presented in Table 3.4. In total, we had 18512 data integrity or access control theorems for theorem proving. We refer to these theorems as verification cases or verification instances. We translated these 18512 cases into different FOL variants in order to evaluate the performance using different provers, heuristics and translations: a total of 74048 FOL theorems. For each of these cases, we ran verification with a time limit of 5 minutes. If the theorem prover did not deduce a result within 5 minutes we treated the result as inconclusive. Given that most verification cases terminate in a few seconds, we believe that this is a reasonable time limit.

6.4.1 FOL Theorem Provers

In these experiments, we used Spass [107] for our unsorted theorem prover. Spass is a FOL theorem prover based on superposition calculus. While Spass supports multiple input formats, we translated the verification cases to Spass's own input format [106]. Spass tries to prove that a conjecture follows from a set of axioms by negating the conjecture and attempting to deduce a contradiction. If this contradiction is found, then the conjecture is proven to follow from the axioms.

Note that Spass supports *soft sorts* [107] which are different than the sorts in many-sorted logic we discussed earlier, and any other sort system we encountered. Soft sorts do not imply mutually exclusive universes. In a soft sort system any universe element may be of a sort, of no sort, or of multiple sorts. Semantically, these sorts are indistinguishable from unary predicates. Furthermore, Spass by default infers soft sorts even if none are explicitly specified. Spass provides a command option that allows us to disable the soft sort system, in which case the theorem prover treats soft sorts as unary predicates. The differences between these soft sorts and sorts as defined in many-sorted logic have been observed before [11]. In the following discussion, whenever we refer to sorts we refer to sorts defined by many-sorted logic. We will use "soft sorts" to refer to Spass's version of sorts specifically.

We used Z3 [25] to evaluate effectiveness of data model verification using many-sorted logic. Z3 is a DPLL(T) [33] based SMT solver that deals with free quantification and uninterpreted functions using E-matching [24].

SMT solvers tend to support many different theories, such as arithmetic, arrays or bit arrays. These theories are combined in propositional logic, which serves to tie the underlying theories without interpreting them. Instead, predicates in underlying theories are treated as propositional variables, and left to the underlying provers to be solved.

Method	# of Timeouts		Verif. Time (sec) Average	Median	Unit Propagations (#) Average	Median	Memory (Mb) Average	Median	Maximum
Spass (Soft sorts on)	3,154	(17.04%)	9.99	9.01	n/a	n/a	60.57	61.55	93.19
Spass (Soft sorts off)	2,878	(15.55%)	11.67	9.21	n/a	n/a	60.46	61.55	111.83
Z3 (Many-sorted)	26	(0.14%)	0.08	0.04	577.01	23	4.02	3.87	285.64
Z3 (Unsorted)	879	(4.75%)	2.28	0.44	1098.34	82	134.59	40.44	15,490.26

Table 6.1: Verification performance summary

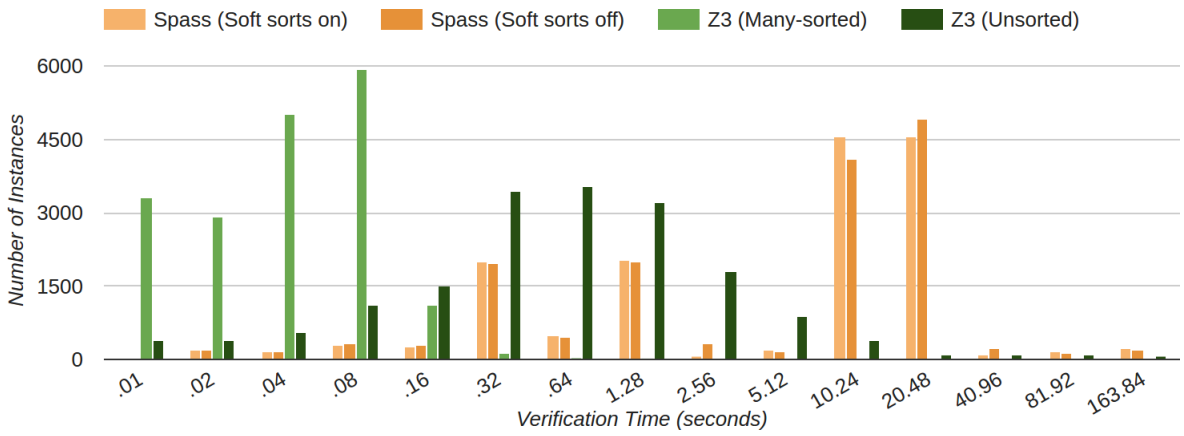


Figure 6.7: Verification time distribution

Partial conclusions from these underlying theories may be propagated to other underlying provers in DPLL(T) in order to reach other conclusions. When used only with free quantification, free sorts and uninterpreted functions (which is denoted as the problem group UF), SMT solvers behave like many-sorted logic theorem provers.

SMT solvers try to find instances that satisfy the specification, so in order to prove that the conjecture follows from axioms, we negate our conjecture and state it as an additional axiom. The conjecture follows from the axioms if and only if this resulting set of axioms is unsatisfiable.

6.4.2 Spass vs. Z3 Performance

Our first set of experiments compare the performance of Spass and Z3 for the purpose of data model verification. These experiments were done solely to detect whether Z3 can sometimes outperform Spass, either by reaching results that Spass could not, or reaching

them in less time. If so, our efforts in translating data models to SMT would increase the performance and/or reduce the ratio of inconclusive results in our data model verification efforts, and therefore increase the viability of data model verification in the real world.

Our results are summarized in Table 6.1 and Figure 6.7. The performance difference was beyond our initial expectations. Note that the Z3 (Unsorted) entries are only relevant for the experiment discussed in the next subsection and can be disregarded for now, as is the case for *Unit Propagations* columns. With soft sorts enabled, Spass produced 3154 inconclusive results (17.04%). With soft sorts disabled, Spass produced 2878 inconclusive results (15.55%). Interestingly, there are 63 cases where enabling sorts led Spass to a conclusive result where disabling sorts did not, yet there are 336 cases where the opposite is true. Performance-wise, Spass performed similarly regardless of the soft sorts setting. For both settings, excluding timeouts, verification took an average of about 10 seconds per case. The median case is just over 9 seconds. Memory consumption averaged at around 60Mb, with the median case of 61.55Mb. Memory consumption peaked at just over 100Mb memory when Spass produced a conclusive result. For inconclusive results, memory consumption peaked at just over 1Gb.

Z3 performed far better than Spass with either heuristic. Z3 produced far fewer inconclusive results, only 26 (0.14%). In addition, in only 3 cases did Z3 fail to produce a result when Spass succeeded. In the remaining 23 cases, neither prover could reach a conclusive result in 300 seconds. On average, Z3 took 0.08 seconds per verification case, with a median time of 0.04 seconds. Spass outperformed Z3 in only 6 cases in terms of time performance, while Z3 outperformed both Spass heuristics in 15269 cases, counting only cases where all provers produced a result. Furthermore, Z3's average memory consumption was just over 4Mb, a with median under 4Mb. Memory consumption peaked at just under 300Mb. However, Z3 tends to consume far more memory when it is failing to produce a conclusive result. In our case, Z3 used 35Gb of memory before forcefully

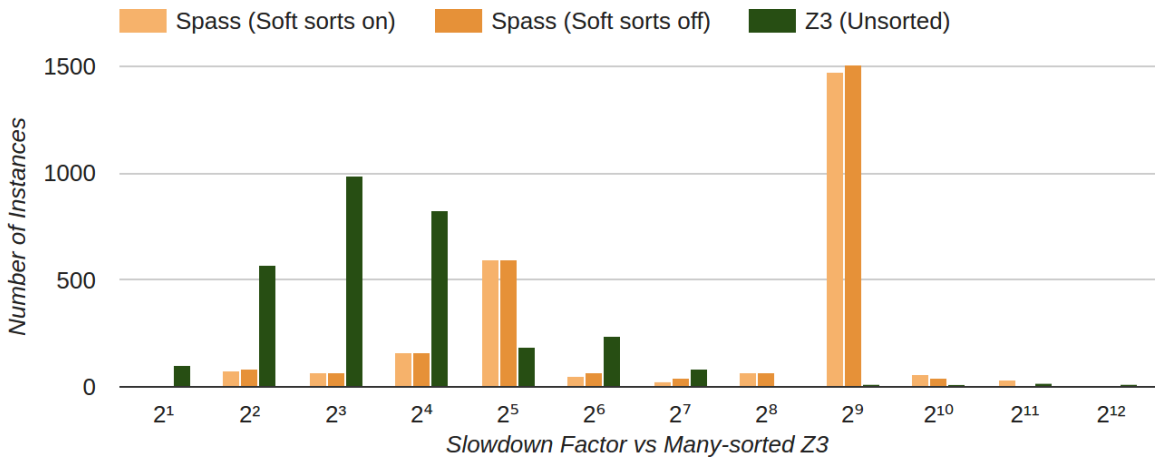


Figure 6.8: Distribution of the slowdown factor compared to (many-sorted) Z3

Method	Average	Median	Interdecile Range
Spass (Soft sorts on)	281.86	185.0	19.5 – 349.5
Spass (Soft sorts off)	299.62	191.0	19.0 – 357.8
Unsorted Z3	80.43	11.8	3.6 – 47.2

Table 6.2: Observed slowdown compared to (many-sorted) Z3

being terminated after 5 minutes.

Figure 6.7 shows the distribution of the verification cases over the verification time ranges for each theorem prover. For example, the leftmost column (labeled .01) shows that Z3 produced a verification result in less or equal than 0.01 seconds 3315 times. Spass achieved a result within this time only 13 times, which is not visible on the chart. The next time range is labeled .02 and shows that Z3 produced a verification result in more than 0.01 seconds but less or equal to 0.02 seconds 2908 times, while Spass with soft sorts on produced a result 208 times within the same timeframe.

We wanted to compare the performance of different provers on case-by-case basis. For each verification case, we calculated the *relative slowdown factor* induced by a prover compared with Z3. So, for example, if an verification case was verified 85 times slower using Spass with sorts enabled when compared with Z3, this counts as a slowdown factor of 85. Figure 6.8 and Table 6.2 summarize this data.

Figure 6.8 contains the distribution of slowdown factors per prover. For example,

Spass (with and without soft sorts) is most frequently 2^9 times slower than Z3. Table 6.2 contains additional information about this slowdown. On average, Z3 was 281.86 times faster than Spass with soft sorts on, and 299.62 times faster with soft sorts off. In the median case, Z3 was 185 and 191 times faster, respectively.

In order to estimate a range of performance increase factor for the majority of cases, we calculated interdecile ranges of these distributions. The interdecile range of a sample is the range of values ignoring the lowest and highest 10% of the sample. It serves to communicate a range of values, ignoring outliers. The interdecile ranges of performance increases of Z3 over Spass with soft sorts on and off are 19.0-349.5 and 19.0-357.8 respectively. This means that, 80% of the time, Spass was one to two orders of magnitude slower than Z3.

In summary, our translation to SMT and use of Z3 for verification increased the performance of verification of our method by two orders of magnitude, and reduced the number of inconclusive results removed all inconclusive results down from around 16% to 0.14%.

6.4.3 Many-sorted vs Unsorted Performance

We observed a drastic improvement in our method's performance by utilizing Z3 instead of Spass. However, this difference was beyond our expectations, and we wanted to investigate the reason behind the performance difference. This is hard to pinpoint since Spass and Z3 are fundamentally different. They utilize a different approach to theorem proving and have different optimizations and heuristics.

During manual investigation of Spass's deduction logs we noticed that Spass was taking a significant amount of time reasoning about types of quantified variables. This is true regardless of whether soft sorts are enabled or not. This reasoning about types would

not be necessary or would be drastically reduced if the theorem prover supported (non-soft) sorts. Even if the model contains a larger number of classes that inherit from one another, causing us to introduce predicates and axioms that resemble the ones generated for unsorted logic, this type reasoning is constrained to a smaller scope of an inheritance cluster instead of the set of all classes.

We implemented an unsorted translation to SMT in order to observe the benefit of using sorts. Because SMT-LIB requires all predicates and functions to be sorted, we defined a single sort (called `Sort`) that we used for all language elements. Since this single sort represents everything, we effectively provide no explicit type information. On top of this sort(less) system we enforce the type system using predicates and axioms using the unsorted translation presented in Section 6.1. Thereby we specify the type system in a way that requires type reasoning in a way that corresponds to the amount of information we provide to Spass.

We ran the same suite of application models and action-invariant pairs using the many-sorted and unsorted translations to SMT. Table 6.1 summarizes the performance of many-sorted and unsorted Z3 verification. Unsorted Z3 did not produce a conclusive result in 879 cases (4.75%). On average, many-sorted Z3 took 0.08 seconds per case whereas unsorted Z3 took 2.28 seconds. Median values are 0.04 for the many-sorted logic and 0.44 for the unsorted translation.

The *Unit Propagations* columns in Table 6.1 refer to the number of DPLL(T) unit propagations done by Z3. On average, the many-sorted translation required 577.01 unit propagations before deducing a conclusive result, with a median number of 23. The unsorted translation required 1098.34 propagations for conclusive results on the average, with a median number of 82. Therefore, Z3 needed to do more work to reach conclusive results when using unsorted logic.

Finally, the memory footprint of verification suffered as well. The many-sorted trans-

lation used an average of 4.02Mb of memory per verification case, with a median of 3.87Mb. The unsorted translation was drastically more demanding, with an average of 134.59Mb and median of 40.44Mb. Memory consumption peaked at over 15Gb of memory for unsorted theorem proving.

Figure 6.7 contains data for the unsorted Z3 translation in addition to (many-sorted) Z3 and Spass results. Similarly, Figure 6.8 and Table 6.2 show the distribution of case-by-case slowdown factors when comparing unsorted Z3 to many-sorted Z3. On average, the many-sorted translation resulted in 80.43 times faster verification compared to the unsorted translation when both methods produced conclusive results. The median case is 11.8, and the interdecile range is 3.6-47.2.

These results imply a large performance difference between many-sorted and unsorted logic verification in Z3. While this does not imply that implementing proper many-sorted logic in Spass would increase Spass's performance by a similar factor, it does indicate that the reduction of reasoning induced by many-sorted over unsorted logic plays a significant role in the performance gain we observed.

6.4.4 Coexecution and Many-Sorted First Order Logic

After having implemented the many-sorted translation, we reevaluated the performance of loop verification using coexecution and sequential iterations. We found that, with a many-sorted translation, the performance benefits to coexecution are less consistent.

We compared coexecution and sequential execution on all actions with loops in applications present in Table 3.4. The results are presented in Table 6.3. We observe gains from using coexecution only for z3 with many-sorted enabled, which is our best performing configuration. In this case, coexecution reduced the number of false positives from

Loop Model	Prover	# of Timeouts / Total	Avg Time (seconds)
Sequential	Spass, Sorts on	284/581 (48.88%)	24.45
	Spass, Sorts off	267/581 (45.96%)	46.11
	Z3	21/581 (3.61%)	1.13
	Z3 Unsorted	59/581 (10.15%)	15.73
Coexecution	Spass, Sorts on	286/581 (49.23%)	25.29
	Spass, Sorts off	263/581 (45.27%)	48.27
	Z3	3/581 (0.52%)	0.87
	Z3 Unsorted	54/581 (9.29%)	8.16

Table 6.3: Coexecution vs sequential execution with many-sorted logic

21 to 3 (3.61% to 0.52%), which is a bigger improvement than the improvements we previously observed. However, in other cases, the differences between coexecution and sequential iteration modeling are not significant.

We believe this difference comes from the differences in our FOL encoding since the coexecution experiments were originally ran. Our current encoding uses type-aware predicates to encode actions. Specifically, we now use a dedicated predicate for each program state and type, as opposed to just state. Because of this, we only define a state predicates specific to a type when necessary because of a state change with regards to said type. As such, for example, if a loop modifies only objects of type A, state predicates for type B will bypass the loop altogether. This is the case even in our unsorted Z3 translation: even though this translation does not utilize sorts, it does have independent state predicates for each type. This makes it easier for the theorem prover to bypass reasoning about loops altogether.

Chapter 7

Related Work

7.1 Modeling and Verification of Web Applications

Nijjar et al. present techniques for analysis and verification of data models in Rails applications [80, 81, 79]. The data model used in Nijjar et al.'s work is a static model that does not represent actions that modify data store states. Properties are checked with respect to association declarations without considering how they are updated via actions. Moreover, some assumptions used in constructing the static data model (such as assuming `belongs_to` associations have exactly one associated object) are not guaranteed to hold by the Rails semantics. In contrast, our model captures the exact behavior of a data store by modeling how actions update the data store states. Their solution does not address the problem of sorts and empty universes, making their verification unsound. Finally, their work does not delve into the difference between logics and their implied encodings.

We checked some of the properties of the Tracks application that were also checked by Nijjar et al. [80, 81, 79]. One of the properties (Every User has a Preference) is proven

to hold by our tool. However this property cannot be proven on the static model used by Nijjar et al. and causes a false positive with their approach. Both approaches are able to prove another invariant (Every Todo has a Context), but in our approach we do not assume that `belongs_to` associations in Rails have exactly one associated object since this assumption is not enforced by the Rails framework.

Near et al. [75] developed Rubicon, a web application verification tool that adds quantification to unit tests and translates tests into verifiable Alloy specifications using symbolic execution. Rubicon uses the Alloy Analyzer for bounded verification of generated specifications. Even though Rubicon is not specifically designed to find access control bugs, they did find one. Since their approach requires the developer to write tests, it requires more effort than our automated method and may miss bugs.

Both Tracks and Fat Free CRM were analyzed by the tool developed by Nijjar et al. and Rubicon. Neither of their tools were able to uncover the four bugs we found. Since the tool developed by Nijjar et al. only analyzes the static data model it is unable to capture the semantics of actions and find the bugs in them. Rubicon's approach, on the other hand, is a testing based framework. Successful tests verify that, after an explicitly stated sequence of steps, the application behaves as expected. Tests are not suitable for verifying that no possible sequence of action executions could lead to a faulty state, which is the condition checked by our verification framework.

Space[77] is a tool for verification of access control in Rails applications. It uses symbolic execution to detect exposures, and checks whether these exposures conform to a set of user supplied patterns. We rely on developer-supplied policies and verify the enforcement, regardless of patterns. Furthermore, our model extraction technique is better suited for extracting models of dynamically generated code.

The Unified Modeling Language (UML) is a language commonly used for specification of object oriented models. The Object Constraint Language (OCL), which is part of the

UML standard, enhances UML with the ability to specify invariants and pre- and post-conditions of methods [82, 104]. Research on verification of OCL specifications have ranged from simulation of object oriented models [90], to interactive verification with automated theorem prover support [4]. However, UML combined with OCL does not provide a way to specify method bodies, whereas method bodies can be mapped to action specifications in our language. Hence, because of the semantic gap between the UML/OCL specifications and actual implementations, the method bodies are unlikely to be modeled precisely using UML/OCL, which means that the bugs we found are likely to be missed by a verification approach based on UML/OCL specifications.

Alloy [62, 63] is formal language for specifying object oriented data models and their properties. Alloy Analyzer is used to verify properties of Alloy specifications. Unlike our work, Alloy focuses on static models and does not directly support specification of actions or dynamic behavior. Moreover, Alloy Analyzer uses SAT-based bounded verification techniques as opposed to the FOL based unbounded verification technique used in our work.

DynAlloy is an extension of Alloy that supports dynamic behavior [42, 43] by translating dynamic specifications onto Alloy. While they talk about *actions* in their work, those actions do not correspond to actions in web applications. Instead, they are more similar to individual statements in programming languages [45]. Their work has focused on verification of data structures, not behaviors in data models of web applications.

There has been prior work on formal modeling of web applications, mainly focusing on state machine based formalisms to capture the navigation behavior[97, 54, 17, 53, 108, 92]. In contrast to these previous efforts, we are focusing on analysis of the data model rather than the navigational aspects of the web applications.

There are previous results on unbounded verification of data-driven web applications based on high level specifications [27, 26, 28]. Deutsch et al. model actions as input/out-

put rules instead of specifying them procedurally, creating a semantic gap between the implementation and the specification of the actions. Due to the semantic gap between the input/output rule format used in their language and the actual implementations of actions, the bugs we found would not be discovered by their verification approach. Additionally, they impose restrictions on the use of quantification in their properties whereas we do not have any restrictions.

Rails's mechanisms for ensuring data validity have been investigated [7], finding potential faults in both how these mechanisms are used and in their implementation inherently. They found that developers rarely used transactions in actions, and often improperly. Our method, thus far, assumes that transactions are used correctly, and as such, we may miss bugs. Enhancing our method to not rely on this assumption is future work.

SafeWeb [58] is a Rails middleware tool for access control. It stands as a layer between the web application and the data. At runtime, it tracks data items and propagates associated permissions in a way that is similar to dynamic taint analysis, raising errors if a user gains access to restricted information. We statically ensure that access control is implemented correctly, while SafeWeb incurs a runtime overhead. In addition, in order to use SafeWeb, existing Rails applications to be fundamentally overhauled. Our goal was to examine existing applications.

RailroadMap [73] is an automated tool for verification of access control in Rails using CanCan and Pundit. The similarities between our approach and theirs end at trying to achieve the same goal. Their program analysis is limited to parsing a few specific Rails files and examining the AST, not even taking file dependencies into account. They expect the Ability class to be declared in a limited manner, not allowing `elsif` branches or branch nesting or method calls in the constructor. They naively assume that not showing url links to actions is sufficient to prevent unrestricted execution of actions. Finally, they

evaluated their method on small applications: all but a few had a single developer and were abandoned in weeks. We could not directly compare our results to theirs because their reported results are not specific enough to compare: they report the number of bugs but no specific description of bugs. In addition, we are confused by them reporting access control bugs in two applications (Artdealer and shiroipantsu) that, as far as we can see, had never in their history of development used access control.

7.2 Access Control

Our basic access control model maps each user, operation and object, to a boolean [70]. There are more elaborate access control systems [38, 59]. Our model can be considered a simplified version of role-based access control (RBAC). Our model of access control reflects the access control mechanisms used in real world Rails gems like CanCan, Can-CanCan and Pundit.

There is an extensive body of research that focuses on verification of access control policies [32, 60], as well as aiding the creation of access control policies [40, 34]. This work describes and investigates policies specified using standards such as XACML. In our work, rather than focusing on policy correctness in isolation, we are focusing on inconsistencies between policy specification and policy enforcement. In most cases such inconsistencies point to bugs in policy enforcement, even though policy specification might itself be deficient. In addition, we focus on verifying access control policies in Rails applications, and XACML is not used in any Rails application we came across.

The problem of access control policy enforcement has been tackled before. For example, there exists work that checks whether a user of a particular role could access a restricted webpage [98]. Their work requires manual specification of user roles and categorizing pages based on which roles can access them. We automatically extract this

information from existing code. Instead of just ensuring authorized actions are accessible, we also ensure that all operations executed by accessing these actions conform to the policy. As an other example, access control can be verified on Java objects, given an appropriate policy [6]. Our problem domain, level of automation, and approach are all different.

7.3 Theorem Prover Based Verification

Verification of software using theorem provers has been explored before in projects such as Boogie [9], Dafny [71] and ESC Java [41]. These projects focus on languages such as C, C#, and Java, and typically require user guidance in the form of explicit pre- and post-conditions, explicit data structure constraints, and loop invariants. While loop invariants may be inferred for certain loops, they are ultimately required to reason about the loops. Our method does not require loop invariants, and uses static analysis to automatically optimize the loop translation to FOL. While low level languages such as C and Java present different challenges then our high level language, we believe that modeling loops via coexecution is applicable and would be beneficial for the verification of loops in low level languages as well.

Another line of related work is inductive verification of abstract data type specifications [51, 74, 46], where algebraic specifications are used to model behaviors of data types such as stacks, queues etc., and automated verification techniques based on term-rewriting systems are used for verification. The types of specifications we focus on, and the verification techniques we use, are significantly different.

The improvements we noticed from using many-sorted first order logic hint at a fundamental mismatch between programming languages and the input of theorem provers. This lack of support for precise reasoning about programming language constructs in the-

orem provers has been noticed and addressed before [23]. Specifically, [23] discusses this problem with regard to ANSI-C basic types and operations, bit-vectors and structures, pointers and pointer arithmetic. They address this problem by devising a theorem prover that supports all these elementary operations. These improvements do not improve on the basic problems with loop verification as tools that use Simplify still require loop invariants [41], and they do not focus on object-oriented code with multiple inheritance.

As part of a research effort to use Spass as the theorem prover engine for interactive theorem proving [11], Spass was modified to support many-sorted logic. This was done in order to make deduction logs sort aware, which in turn makes it possible to reconstruct readable proofs from these logs and show them to the user for the purpose of interactive theorem proving. They observed an increase in the number of theories Spass could solve. However, this modification was done for performance reasons, making it reasonable to expect an even larger performance gain from sorts in Spass. The source of this Spass modification is not available, and so we could not include it as part of our experiments.

There are other theorem provers that can be used for data model verification. Vampire [68] is a high performance FOL theorem prover that supports sorts. Snark [96] is another FOL theorem prover, also supporting sorts. We plan to, as part of our future work, implement automatic translation of data models into TPTP syntax [101, 102]), the syntax of the test suite that is used by the annual World Championship for Automated Theorem Proving [100, 85]. This language is readable by many theorem provers, including Spass and Z3. However, given that many-sorted logic has only recently been added to TPTP [99], we expect that the highest performing theorem provers are optimized for unsorted logic. Unless the theorem prover integrates sorts within its resolution engine, we can expect many-sorted logic to perform no better than unsorted logic. Support for many-sorted logic is possible to implement syntactically (e.g., by treating sorts as predicates and implicitly introducing axioms that define disjoint universes), however, this

would not result in the performance gains we observed.

In addition to unsorted and many-sorted logic, there exists order-sorted logic [48]. Order-sorted logic defines a partially ordered set of sorts, and the universes that correspond to these sorts are such that universe of class c_1 is a subset of the universe of class c_2 if $c_1 \leq c_2$. While order-sorted logic is highly similar to our data-models involving multiple inheritance, we are not aware of theorem provers that support it in first order logic with free quantification.

7.4 Coexecution

An interesting parallel can be drawn between coexecution of loop iterations and snapshot isolation in the domain of databases [10, 37]. The coexecutability problem is similar to the problem of equivalence of serializability and snapshot isolation. However, we see no parallel between our delta union and the delta apply operations and snapshot isolation notions such as *first-committer-wins*, transactions aborting or committing based on conflicts etc. Our purpose is verification viability, not scalability or optimization of transactions. Our domain of application is quite different, with our model being purely based on sets and relations with no basic types, and operations being only creates and deletes (which we differentiate in our approach instead of considering both of them as writes.

There exists a long body of work focusing on operation commutativity with applications such as automatically parallelizing data structures [66] and computation [61, 91]. Automatic loop parallelization has been researched for decades [5, 8, 52]. This prior research acknowledges loop dependencies as problematic for parallelization, and the potential for performance increase if no such dependencies exist. While we are also avoiding loop dependencies, our purpose is not optimization or making execution scalable, but

making verification more feasible in practice. Coexecution is a theoretical concept that is not executable in actual hardware. Furthermore, there exist parallelizable loops that are not coexecutable.

Semantic properties of operations have been used for the purposes of simplifying verification [35]. This is similar to our approach at a high level. However, we do static analysis of a particular condition that allows us to use a completely alternate definition of a loop, whereas [35] iteratively abstracts and subsequently reduces the model in order to infer and enhance atomicity rules without altering the validity of the given invariants. Their problem, domain of application, goal and solution are fundamentally different.

Semantic properties of operations have been used for the purposes of simplifying verification [35]. This is similar to our approach at a high level. However, we do static analysis of a particular condition that allows us to use a completely alternate definition of a loop, whereas [35] iteratively abstracts and subsequently reduces the model in order to infer and enhance atomicity rules without altering the validity of the given invariants. Their problem, domain of application, goal and solution are fundamentally different.

7.5 Extraction

Rubicon [75] uses symbolic execution for program analysis. Their symbolic execution is fully explained in a technical report [76]. Like them, we use the dynamic features of an unmodified Ruby runtime to override concrete methods with their symbolic counterparts. However, they override methods with their symbolic counterparts only once, before symbolic execution has started. Considering that they use classical symbolic execution in a Ruby interpreter, there are some key differences between our methods: most importantly, in the fact that we can extract models from dynamically generated source code. Recently, they used their symbolic execution technique to extract access control signa-

tures from Rails programs [77]. Their experimental set is limited to applications much smaller and simpler than ours. We suspect this is caused by their extraction method not supporting dynamic features as ours, so they had to exclude these applications from their experimental set.

There has been work on the static analysis of Rails [56]. This line of work focuses on typechecking Rails applications and builds on DRuby [44], which is a Ruby static type-checker. The presented techniques infer types and detect errors by converting each statement into a type constraint, and exhaustively applying a set of rewrite rules. In effect, they insert type information that refers to dynamically generated methods by assuming that dynamically generated methods will adhere to the core Rails specification. As such, their method, as our model extraction, relies on dynamic method generation being input-independent. However, instead of investigating dynamically generated methods, they have assumptions about the semantics of these methods that are based on Rails specifications. However, they admit that their assumptions about the functionality of Rails methods may be incomplete or incorrect. Their approach not handle methods generated by a third party library, as those were not explicitly listed by their tool. In addition, if a third party library augments a core Rails method, their approach may assume the wrong semantics.

RubyX [21] is a tool for symbolic execution in Rails that can be used to find access control bugs. It uses manually written scripts, each of which has to setup a database with symbolic values, execute an action, manually capture relevant output of the action, and check whether specific post-conditions hold. We require no manual effort from the developer both in terms of specifying expectations of correctness and scenarios under which these expectations should be met. Furthermore, symbolic model extraction does not rely on SMT solvers and a custom symbolic runtime. Because we do not use SMT solvers during model extraction, we are not limited to conditionals that can be specified

in decidable logic fragments. We accomplish model extraction without a custom runtime that keeps track of symbolic values. Furthermore, they use DRails [56] to make specific usages of Rails code explicit, whereas we capture metaprogramming natively.

Symbolic execution [67, 55, 65, 47, 93, 18] is a well know technique for program analysis. Instead of executing source code in a normal runtime, symbolic execution will execute source code in an alternate runtime, operating on symbolic values instead of concrete values. These symbolic values are abstractions of concrete values. SAT and SMT solvers are used in branch conditions to determine if branch conditions are satisfiable, in order to guide path exploration for the purpose of testing. We use an unmodified Ruby runtime which makes our technique easier to implement, and we do not use solvers to resolve branch conditions as our purpose does not extend beyond extracting the model of a branch condition.

Concolic execution [93] extends on symbolic execution by keeping track of concrete values as well as symbolic. This is useful when solvers are not able to check satisfiability or find satisfying assignments to a branch conditions. One could look at our treatment of dynamic features as concolic, as we execute them concretely instead of symbolically.

Rubydust [57] attempts to typecheck Ruby code, accomplishing this by wrapping objects with type constraints and running actual code. They also use some basic instrumentation. This makes their approach similar to the core idea behind our extraction by instrumented execution technique, and in both cases, the goal is to get around dynamic features of Ruby. However, their wrapped objects contain type information whereas we inject abstract syntax trees into ActiveRecord objects and variables. Their solution focuses on general purpose Ruby which our solution cannot cover, but within our domain and with the code generation purpose, we cover much larger applications.

Chapter 8

Conclusion

In this dissertation we presented an approach for verification of data models of web applications. Specifically, our automated method is effective in finding data integrity bugs and access control bugs in web applications.

Verification of data integrity and access control of web applications involves several steps. First, to extract an abstract data store model from a given application, we developed *symbolic extraction*, an approach for model extraction that can handle dynamically generated code. This was necessary considering the dynamic nature of Ruby and typical Rails programming.

We designed the *Abstract Data Store* (ADS) modeling language to represent an abstraction of a web application that focuses on how an application stores, manages and accesses its data. This model is designed to be precise enough to find bugs with a low rate of false positives, yet abstract enough to be translatable to first order logic and verifiable using theorem provers quickly and effectively.

Finally, we translate the abstract data store, including correctness criteria to first order logic, to be verified using off the shelf theorem provers. To improve verification

viability, we found two ways to improve our translation: using coexecution to model loops, and leveraging many-sorted first order logic. Coexecution allows us to model loops in a way that does not directly correspond to a potential execution on a computer, but is significantly easier to reason about by theorem provers, and is logically equivalent under conditions we defined. Many-sorted logic lets us define our model with fewer axioms, lessening the theorem prover's burden of reasoning about the type system. We experimentally demonstrate that these improvements highly increase the viability of verification.

We experimentally evaluated our approach on 19 open source Ruby on Rails web applications. Using this approach we identified numerous bugs regarding both data integrity and access control.

Future work. Our work can be improved in two general ways: improving the quality of verification on the current problem domain, and expanding the method to other domains.

Both model extraction and translation to logic can be improved. Model extraction can be made more precise to avoid false positives: for example, by differentiating objects that are saved in actions from those that are not. Furthermore, our implementation is aimed at specific versions of Rails and associated libraries. This set can be expanded.

As for the translation to first order logic, we believe that some real world abstract data store models can be translated to decidable fragments of first order logic. We do not know what the conditions required to guarantee translation to a decidable fragment are, or how restrictive they would be, but this would definitely improve our verification performance. In addition, we believe that we could greatly speed up translation and verification if we could trim the schema on a per-action or per-property basis in order to further reduce the number of axioms used to model a behavior.

In addition, we would like to expand the domain of our approach. First, we would

like to see how our method fairs with other web application frameworks, such as Django or Spring. We believe that our approach could be useful to them, given a lack of research on data models for those application frameworks.

Second, we would like to expand on abstract data stores to cover basic type field manipulation. While this would probably hurt the viability of verification using first order logic, SMT is a viable alternative. If successful, this would make our approach more precise and potentially applicable to other domains.

Third, we would like to investigate applications that use non-relational databases. While this would probably require fundamental changes to the abstract data store modeling language, data integrity bugs are far more likely in these applications.

Finally, we would like to develop a new programming language based on abstract data stores that is simple to use and expressive enough to synthesize portions of web applications. In addition to providing ease of use to the developer, data models written in this language would be verifiable using our approach, making web application development easier and resulting applications more robust and secure.

Bibliography

[1] ekremkaraca/awesome-rails: A collection / list of awesome projects, sites made with Rails.

[2] Health Insurance Marketplace — Healthcare.giv. https://www.healthcare.gov/, 2015.

[3] activeadmin/activeadmin: The administration framework for Ruby on Rails applications., Aug. 2016. `https://github.com/activeadmin/activeadmin`.

[4] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hahnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, 2005.

[5] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (SIGPLAN 1988)*, PLDI '88, pages 308–317, New York, NY, USA, 1988. ACM.

[6] A. Ali and M. Fernández. Static enforcement of role-based access control. In M. H. ter Beek and A. Ravara, editors, *Proceedings 10th International Workshop on Automated Specification and Verification of Web Systems, WWV 2014, Vienna, Austria, July 18, 2014.*, volume 163 of *EPTCS*, pages 36–50, 2014.

[7] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1327–1342, New York, NY, USA, 2015. ACM.

[8] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE (1993)*, 81(2):211–243, 1993.

[9] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO 2005)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[10] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM International Conference on Management of Data (SIGMOD 1995)*, pages 1–10, 1995.

[11] J. C. Blanchette, A. Popescu, D. Wand, and C. Weidenbach. More SPASS with isabelle - superposition with hard sorts and configurable simplification. In *Interactive Theorem Proving - Third International Conference, (ITP 2012), Princeton, NJ, USA, August 13-15, 2012. Proceedings*, pages 345–360, 2012.

[12] I. Bocic and T. Bultan. Inductive verification of data model invariants for web applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, May 2014.

[13] I. Bocic and T. Bultan. Coexecutability for efficient verification of data model updates. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 744–754, 2015.

[14] I. Bocic and T. Bultan. Data model bugs. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 393–399, 2015.

[15] I. Bocic and T. Bultan. Efficient data model verification with many-sorted logic. In *30th IEEE/ACM International Conference on Automated Software Engineering ASE 2015, Lincoln, Nebraska, USA, November 9-13, 2015*, 2015.

[16] I. Bocic and T. Bultan. Finding access control bugs in web applications with cancheck. In *31st IEEE/ACM International Conference on Automated Software Engineering ASE 2016, Singapore*, 2016.

[17] M. Book and V. Gruhn. Modeling web-based dialog flows for automatic dialog control. In *Proceedings of the 24th IEEE/ACM International Conference Automated Software Engineering (ASE 2004)*, pages 100–109, 2004.

[18] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI 2008)*, pages 209–224, 2008.

[19] cancan — RubyGems.org — your community gem host, Sept. 2013. `http://rubygems.org/gems/cancan`.

[20] ryanb/cancan • GitHub, Nov. 2015. `https://github.com/ryanb/cancan`.

[21] A. Chaudhuri and J. S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 585–594, 2010.

[22] K. Claessen, A. Lilliestr\"om, and N. Smallbone. Sort it out with monotonicity - translating between many-sorted and unsorted first-order logic. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 207–221, 2011.

[23] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *Proceedings of the 17th International Conference on Computer Aided Verification, (CAV 2005)*, pages 296–300, 2005.

[24] L. de Moura and N. Bjrner. Efficient e-matching for smt solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.

[25] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[26] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *Journal of Computer and System Sciences*, 73(3):442–474, 2007.

[27] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. A system for specification and verification of interactive, data-driven web applications. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *SIGMOD Conference*, pages 772–774. ACM, 2006.

[28] A. Deutsch and V. Vianu. WAVE: Automatic verification of data-driven web services. *IEEE Data Engineering Bulletin*, 31(3):35–39, 2008.

[29] devise — RubyGems.org — your community gem host, Sept. 2013. `http://rubygems.org/gems/devise`.

[30] Discourse, Mar. 2014. `www.discourse.org`.

[31] The Web framework for perfectionists with deadlines — Django, Feb. 2013. `http://www.djangoproject.com`.

[32] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[33] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 81–94, 2006.

[34] S. Egelman, A. Oates, and S. Krishnamurthi. Oops, i did it again: Mitigating repeated access control errors on facebook. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2295–2304, New York, NY, USA, 2011. ACM.

[35] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, pages 2–15, 2009.

[36] Fat Free CRM - Ruby on Rails-based open source CRM platform, Sept. 2013. `http://www.fatfreecrm.com`.

[37] A. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[38] D. Ferraiolo and R. Kuhn. Role-based access controls. *Proc. of 15th NIST-NSA National Computer Security Conference*, 1992.

[39] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[40] K. Fisler and S. Krishnamurthi. A model of triangulating environments for policy authoring. In J. B. D. Joshi and B. Carminati, editors, *SACMAT 2010, 15th ACM Symposium on Access Control Models and Technologies, Pittsburgh, Pennsylvania, USA, June 9-11, 2010, Proceedings*, pages 3–12. ACM, 2010.

[41] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[42] M. F. Frias, J. P. Galeotti, C. L. Pombo, and N. Aguirre. Dynalloy: upgrading alloy with actions. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 442–451, 2005.

[43] M. F. Frias, C. L. Pombo, J. P. Galeotti, and N. Aguirre. Efficient analysis of dynalloy specifications. *ACM Transactions on Software Enginnering Methodology*, 17(1), 2007.

[44] M. Furr, J. hoon (David) An, J. S. Foster, and M. W. Hicks. Static type inference for ruby. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2009)*, pages 1859–1866, 2009.

[45] J. P. Galeotti and M. F. Frias. Dynalloy as a formal method for the analysis of java programs. In *Software Engineering Techniques: Design for Quality, SET 2006, October 17-20, 2006, Warsaw, Poland*, pages 249–260, 2006.

[46] S. J. Garland and J. V. Guttag. Inductive methods for reasoning about abstract data types. In J. Ferrante and P. Mager, editors, *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL 1988)*, pages 219–228. ACM Press, 1988.

[47] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 05)*, pages 213–223, 2005.

[48] J. A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.

[49] Google Health. http://health.google.com/.

[50] Google Powermeter. http://www.google.org/powermeter/.

[51] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, 1978.

[52] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[53] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proceedings of the 25th IEEE/ACM Int. Conf. Automated Software Engineering (ASE 2010)*, pages 235–244, 2010.

[54] M. Han and C. Hofmeister. Relating navigation and request routing models in web applications. In *Proceedings of the 10th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS 2007)*, pages 346–359, 2007.

[55] S. L. Hantler and J. C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, September 1976.

[56] J. hoon (David) An, A. Chaudhuri, and J. S. Foster. Static typing for ruby on rails. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 590–594, 2009.

[57] J. hoon (David) An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for ruby. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, pages 459–472, 2011.

[58] P. Hosek, M. Migliavacca, I. Papagiannis, D. M. Eyers, D. Evans, B. Shand, J. Bacon, and P. Pietzuch. Safeweb: A middleware for securing ruby-based web applications. In *Middleware 2011 - ACM/IFIP/USENIX 12th International Middleware Conference, Lisbon, Portugal, December 12-16, 2011. Proceedings*, pages 491–511, 2011.

[59] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to attribute based access control (abac) definition and considerations. *NIST Special Publication*, 800:162, 2014.

[60] G. Hughes and T. Bultan. Automated verification of access control policies using a SAT solver. *STTT*, 10(6):503–520, 2008.

[61] O. H. Ibarra, P. C. Diniz, and M. C. Rinard. On the complexity of commutativity analysis. *International Journal of Foundation of Computer Science*, 8(1):81–94, 1997.

[62] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Enginnering and Methodology (TOSEM 2002)*, 11(2):256–290, 2002.

[63] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Massachusetts, 2006.

[64] kandanapp/kandan, Sept. 2013. `http://github.com/kandanapp/kandan`.

[65] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[66] D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, pages 528–541, 2011.

[67] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[68] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013), Saint Petersburg, Russia, July 13-19, 2013.*, pages 1–35, 2013.

[69] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming (JOOP 1988)*, 1(3):26–49, Aug. 1988.

[70] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, Jan. 1974.

[71] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 348–370, 2010.

[72] Lobsters, Mar. 2014. `https://lobste.rs`.

[73] S. Munetoh and N. Yoshioka. Model-assisted access control implementation for code-centric ruby-on-rails web application development. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, pages 350–359, 2013.

[74] D. R. Musser. On proving inductive properties of abstract data types. In *Proceedings of the 7th ACM Symp. Principles of Programming Languages (POPL 1980)*, pages 154–162, 1980.

[75] J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In *Proceedings of the ACM SIGSOFT 20th Int. Symp. Foundations of Software Engineering (FSE 2012)*, pages 60:1–60:11, 2012.

[76] J. P. Near and D. Jackson. Symbolic execution for (almost) free: Hijacking an existing implementation to perform symbolic execution. Technical Report MIT-CSAIL-TR-2014-007, MIT, April 2014.

[77] J. P. Near and D. Jackson. Finding security bugs in web applications using a catalog of access control patterns. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 947–958, 2016.

[78] J. Nijjar. *Analysis and Verification of Web Application Data Models*. PhD thesis, University of California, Santa Barbara, Jan. 2014.

[79] J. Nijjar, I. Bocić, and T. Bultan. An integrated data model verifier with property templates. In *Proceedings of the ICSE Workshop on Formal Methods in Software Engineering (FormaliSE 2013)*, 2013.

[80] J. Nijjar and T. Bultan. Bounded verification of Ruby on Rails data models. In *Proceedings of the 20th Int. Symp. on Software Testing and Analysis (ISSTA 2011)*, pages 67–77, 2011.

[81] J. Nijjar and T. Bultan. Unbounded data model verification using SMT solvers. In *Proceedings of the 27th IEEE/ACM Int. Conf. Automated Software Engineering (ASE 2012)*, pages 210–219, 2012.

[82] OMG unified modeling language specification, version 1.3. http://www.omg.org.

[83] Open Source Rails, Jan. 2016. `http://www.opensourcerails.com`.

[84] ActsAsParanoid/acts_as_paranoid: ActiveRecord plugin allowing you to hide and restore records without actually deleting them., Aug. 2016. https://github.com/ActsAsParanoid/acts_as_paranoid.

[85] F. Pelletier, G. Sutcliffe, and C. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.

[86] W. V. Quine. Quantification and the empty domain. *J. Symb. Log.*, 19(3):177–179, 1954.

[87] Ruby on Rails, Feb. 2013. http://rubyonrails.org.

[88] Rails Routing from the Outside In - Ruby on Rails Guides, Jan. 2016. guides.rubyonrails.org/routing.html#crud-verbs-and-actions.

[89] Overview - Redmine, Sept. 2014. www.redmine.org.

[90] M. Richters and M. Gogolla. Validating UML models and OCL constraints. In *Proceedings of the 3rd Int. Conf. Unified Modeling Language (UML 2000), LNCS 1939*, 2000.

[91] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS 1997)*, 19(6):942–991, 1997.

[92] E. D. Sciascio, F. M. Donini, M. Mongiello, R. Totaro, and D. Castelluccia. Design verification of web applications using symbolic model checking. In *Proceedings of the 5th Int. Conf. Web Engineering (ICWE 2005)*, pages 69–74, 2005.

[93] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 05)*, pages 263–272, 2005.

[94] Spring Framework — SpringSource.org, Feb. 2013. http://www.springsource.org.

[95] macfanatic/SprintApp, Sept. 2014. https://github.com/macfanatic/SprintApp.

[96] M. E. Stickel, R. J. Waldinger, M. R. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, pages 341–355, 1994.

[97] P. D. Stotts, R. Furuta, and C. R. Cabarrus. Hyperdocuments as automata: Verification of trace-based browsing properties by model checking. *ACM Transactions on Information Systems (TOIS 1998)*, 16(1):1–30, 1998.

[98] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[99] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP typed first-order form with arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, pages 406–419, 2012.

[100] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[101] G. Sutcliffe, C. B. Suttner, and T. Yemenis. The TPTP problem library. In *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, pages 252–266, 1994.

[102] TPTP Syntax, Jan. 2015. `http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html`.

[103] Tracks, Sept. 2013. `http://getontracks.org`.

[104] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[105] https://petitions.whitehouse.gov/, 2015.

[106] C. Weidenbach. Spass input syntax version 1.5. http://www.spass-prover.org/download/binaries/spass-input-syntax15.pdf.

[107] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *Proceedings of the 22nd Int. Conf. Automated Deduction (CADE 2009), LNCS 5663*, pages 140–145, 2009.

[108] S. Yuen, K. Kato, D. Kato, , and K. Agusa. Web automata: A behavioral model of web applications based on the MVC model. *Information and Media Technologies*, 1(1):66–79, 2006.